TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Rechnernetze

Master-Arbeit

KONZEPT ZUR UNTERSTÜTZUNG VON ENTWICKLERN BEI DER AUFGABENBEARBEITUNG IN EINER TESTGETRIEBENEN CROWDSOURCING PLATTFORM FÜR SOFTWAREENTWICKLUNG

Philipp Dienst
Geboren am: 9. Februar 1991 in Dresden
Matrikelnummer: 3657242
Immatrikulationsjahr: 2010

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

Betreuer
Dr. Tenshi Hara
Dr. Iris Braun

Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Eingereicht am: 22. August 2017

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Professur Rechnernetze

Aufgabenstellung für die Master-Arbeit

THEMA: Konzept zur Unterstützung von Entwicklern bei der Aufgabenbearbeitung in einer testgetriebenen Crowdsourcing Plattform für Softwareentwicklung

Name:	Dienst, Philipp	Studiengang:	Master Medieninf. (PO 2010)
Matrikel-Nummer:	3657242	Projekt/Fokus:	Service and Cloud Computing
verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill		
involvierte Mitarbeit	Dr.-Ing. Tenshi Hara, Dr.-Ing. Iris Braun		
Beginn am:	3.2.2017	einreichen bis:	14.7.2017

ZIELSTELLUNG

Das Outsourcing von Teilen des Softwareentwicklungsprozesses ist verbreitet. Häufig suchen Firmen Aushilfen in Freelancer Portalen oder lagern größere Aufgaben an spezialisierte Unternehmen aus. Ein relativ neuer Ansatz sind Crowdsourcing Plattformen für die Softwareentwicklung. Der Auftraggeber kann auf solchen Portalen seine Aufgaben und Wünsche einer breiten Masse an Programmierern zugänglich machen. Dabei hängt die Qualität sowie der RoI stark von der Dokumentation und Aufgabenstellung ab, die den beauftragten IT-Dienstleistern vorgelegt werden. Die Einstiegshürde für externe Entwickler kann dabei herausfordernd sein, vor allem der Einstieg in ein neues Softwareprojekt ist ein aufwendiger Prozess. Wird dem Entwickler eine Aufgabe innerhalb des Projektes zugeteilt, müssen zunächst Domäne und Technologie sowie bestehende Implementationen analysiert werden. Hierfür stehen meist nur Spezifikationen und bereits vorhandener Quellcode zur Verfügung.

Im Rahmen der Masterarbeit soll ein Konzept zur Unterstützung der Softwareentwickler bei diesem Prozess entstehen. Den Kontext dieser Arbeit bildet dabei eine Crowdsourcing Plattform für Softwareentwicklung, in der ein Entwickler eine Aufgabe innerhalb eines Softwareprojekts lösen soll. In diesem Kontext stehen ihm zunächst natürlich-sprachliche Tests und eine Aufgabenbeschreibung zur Verfügung.

Neben einer State-of-the-Art-Analyse soll geklärt werden, welche Informationen ein Entwickler benötigt, um sich in einem für ihn unbekannten Softwareprojekt zurecht zu finden. Im Anschluss soll untersucht werden, wie diese Informationen visuell aufbereitet werden müssen, um verständlich und interaktiv nutzbar zu sein.

Aufbauend auf den Ergebnissen der State-of-the-Art-Analyse soll ein Webinterface konzipiert werden, welches Entwicklern die Bearbeitung von Programmieraufgaben innerhalb eines Softwareprojektes ermöglicht.

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(verantwortlicher Hochschullehrer)

SCHWERPUNKTE

- Analyse von bestehenden Konzepten zur Unterstützung von Entwicklern, bei der Bearbeitung von Aufgaben innerhalb eines Softwareprojektes,
- Definieren von Anforderungen,
- Definieren von Bewertungskriterien für die Anforderungserfüllung,
- Konzeption einer Lösung im Kontext einer Crowdsourcing Plattform für Softwareentwicklung,
- Prototypische Umsetzung des Konzepts als Webinterface,
- Evaluation des Konzepts anhand ausgewählter Beispielaufgaben, und
- Evaluation und Auswertung der Ergebnisse.

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Masterarbeit mit dem Titel **„Konzept zur Unterstützung von Entwicklern bei der Aufgabenbearbeitung in einer testgetriebenen Crowdsourcing Plattform für Softwareentwicklung“** selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle wörtlich oder sinngemäß übernommenen Gedanken und Zitate als solche kenntlich gemacht habe. Ich erkläre ferner, dass ich die vorliegende Arbeit an keiner anderen Stelle als Prüfungsarbeit eingereicht habe oder einreichen werde.

Philipp Dienst
Dresden, 21.08.2017

INHALTSVERZEICHNIS

1. EINLEITUNG

Im einführenden Kapitel soll zunächst das Problemfeld sowie der zugehörige Kontext vorgestellt werden. Darauf aufbauend wird die Zielsetzung dieser Arbeit beschrieben und die methodische Vorgehensweise anhand des konkreten Aufbaus der Arbeit geschildert.

1.1. EINFÜHRUNG

Im Umfeld von Softwareentwicklungsprojekten ist das Outsourcing von Teilen des Prozesses längst verbreitet und eine häufig genutzte Vorgehensweise. Grund hierfür sind beispielsweise der kompetitive Kosten- und Leistungsdruck sowie der Wunsch nach einem skalierbaren Entwicklerteam [Kra16]. Egal, welche Form des Outsourcings von Softwareentwicklung betrachtet wird, die Qualität und der Return of Invest (RoI) hängen immer stark von der Kommunikation, Dokumentation und Aufgabenstellung ab.
Eine Form, bei der diese Merkmale besonders wichtig sind, ist das *Crowdsourcing*. Der Begriff beschreibt die Möglichkeit von Unternehmen, mittels Technologien des Web 2.0 die bislang an kommerzielle Auftragnehmer ausgelagerten Aufgaben einer anonyme Gruppe (*crowd*) zugänglich machen zu können [Bäc]. Der Begriff Web 2.0 beschreibt dabei Internettechniken und -dienste, welche desktopähnliche Internetanwendungen (*rich internet applications*) bereitstellen, um Generierung und Austausch von Inhalten und Wissen zwischen Internetnutzern zu ermöglichen [Bäc]. Im Kontext Crowdsourcing werden dabei häufig Web-Plattformen genutzt, um Aufgaben einer breiten Masse zugänglich machen zu können. Ein etablierter und erfolgreicher Vertreter dieser Form des Outsourcings ist die Plattform *TopCoder*[1]. Diese ermöglicht es Unternehmen, Teile ihres Softwareentwicklungsprozesses an eine *Crowd* von Softwareentwicklern zu vermitteln. Die auszulagernden Entwicklungsaufgaben werden hierfür von TopCoder Mitarbeitern, welche Crowdsourcingprozess die Rolle des *Task-Distributor* einnehmen, wiederum in kleinere Teilaufgaben aufgeteilt und anschließend auf der Plattform ausgeschrieben. Nimmt ein Entwickler (*Crowdworker*) die Aufgabe an, erhält er eine Beschreibung und meist ein Code-Grundgerüst.

[1] https://www.topcoder.com (1.3.2017)

1.2. PROBLEMSTELLUNG

Für den Crowd-Entwickler ist neben dem zeitaufwendigen manuellen Aufsetzen des Projekts, das Einarbeiten in den vorhandenen Quellcode ein aufwendiger Prozess. Ähnlich wie beim Einstieg in ein bestehendes Softwareprojekt, stellt dieser Vorgang eine Einstiegshürde für den Entwickler dar. Anders als beim Eintritt in ein Projekt eines Softwareteams, muss der Crowdworker die Einarbeitung meist alleine vollziehen. Die Performance und Qualität seiner Lösung hängen dabei stark von der gegebenen Aufbereitung und Spezifikation der Aufgabe ab [LS13]. Die Aufgabenabwicklung bzw. Durchführung der Programmieraufgaben findet meist auf den lokalen Systemen der Crowdworker statt, was die Kommunikation mit dem Task-Distributor erschwert. Bei fehlenden Informationen zur Aufgabe oder aufkommenden Fragen ist meist nur die Nutzung asynchroner Kommunikationsformen möglich.
Ein weiteres Problem von Crowdsourcing-Plattformen ist der langwierige Evaluationsprozess. Hierfür muss der Crowdworker zunächst die lokal bearbeitete Aufgabe auf die Plattform laden und auf eine Softwareevaluation des Task-Distributor warten.

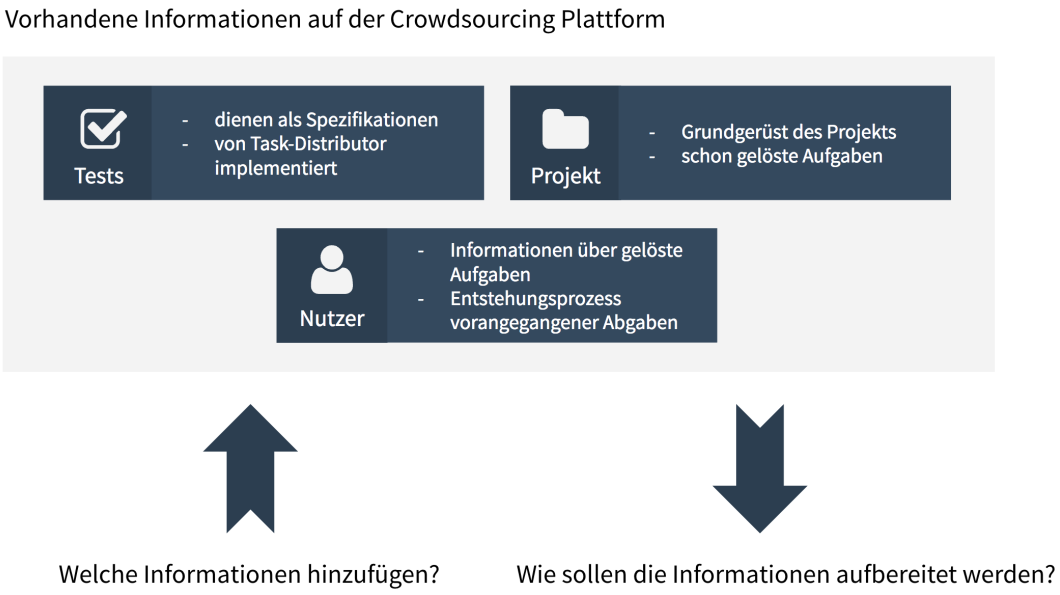


Abbildung 1.1.: Problem/Fragestellung dieser Arbeit

Aufgrund der genannten Probleme soll im Rahmen dieser Arbeit ein Konzept für eine testgetriebene Crowdsourcing-Plattform für Softwareentwicklung betrachtet werden, welche eine direkte Abgabe ermöglicht (siehe Abschnitt 4.1). Um dies dem Crowdworker anbieten zu können, folgen die Task-Distributor dem Ansatz der testgetrieben Entwicklung (TDD) [Per10]. Hierfür erarbeiten die Task-Distributoren zunächst eine Grundstruktur des Projekts und implementieren Softwaretests für die auszulagernden Aufgaben. Die Größe der Aufgabe soll möglichst klein sein, um die auszulagernde Funktionalität mit Unit-Tests abdecken zu können. Im Anschluss werden die Aufgaben auf die Plattform gestellt, in welcher sie implementiert werden müssen. Hierfür soll die Plattform eine an den Crowdsourcingprozess angepasste Entwicklungsumgebung anbieten, welche den Entwickler bei der Lösung maximal unterstützt. Daraus ergeben sich zwei Fragen für die Entwicklung einer solchen Komponente, welche in Abbildung 1.1 dargestellt werden.

1.3. ZIELSETZUNG

Ziel dieser Arbeit ist es, den Aufwand für die Einarbeitung und Bearbeitung einer ausgeschriebenen Softwareentwicklungsaufgabe zu senken. Dabei muss zunächst analysiert werden, welche Informationen ein Entwickler für die Lösung einer Aufgabe benötigt. Auf Grundlage des Konzepts einer testgetriebenen Crowdsourcing-Plattform für Softwareentwicklung soll ein Webinterface umgesetzt werden, welches den Nutzer bei der Bearbeitung von Programmieraufgaben unterstützt. Hierfür soll der Entwickler zusätzliche aufgaben- und domänenspezifische Informationen zur Verfügung gestellt bekommen, die ihm einen einfacheren Einstieg in ein für ihn unbekanntes Softwareprojekt ermöglichen. Weiterhin soll die kognitive Belastung und der Navigationsaufwand für den Entwickler gesenkt werden.

1.4. STRUKTUR DER ARBEIT

Die vorliegende Arbeit gliedert sich in sieben Kapitel. Im Anschluss an die Einleitung folgt die Anforderungsanalyse. Dabei wird zunächst in das Thema Programmverstehen eingeführt und schon existierende Anforderungen an ähnliche Konzepte vorgestellt. Auf Grundlager dieser beiden Abschnitte werden anschließend die Anforderungen an diese Arbeit aufgestellt. Im dritten Kapitel werden für die Umsetzung benötigte Technologien in einer State-of-the-Art Analyse vorgestellt. Weiterhin werden ähnliche Konzepte für Crowd-basierte Softwareentwicklung betrachtet und bewertet. Im vierten Kapitel wird der Kontext dieser Arbeit in Form der polylith-IDE eingeführt. Im Anschluss wird das Konzept dieser Arbeit komponentenweise dargestellt. Im darauffolgenden Kapitel wird die Umsetzung des erarbeiteten Konzepts in Form eines *Proof of Concept* beschrieben. Dabei wird auf die zugrundeliegende Architektur, als auch auf das Vorgehen bei der Implementierung eingegangen. Im sechsten Kapitel wird der Prototyp in einer formativen Nutzerevaluation bewertet und das Ergebnis präsentiert. Im siebten Kapitel wird eine Zusammenfassung und Einschätzung des erarbeiteten Konzepts gegeben. Abschließend werden das weitere Vorgehen und Anknüpfungspunkte für folgende Arbeiten aufgezeigt.

2. ANFORDERUNGSANALYSE

Um Programmierer bei der Lösung und Bearbeitung einer abgeschlossenen Aufgabe innerhalb eines Softwareprojekts zu unterstützen, erfordert es zunächst ein grundlegendes Verständnis, wie Entwickler Programme verstehen. Erst auf Grundlage dieser Erkenntnisse lassen sich verständnisfördernde Strukturen und Hilfen erarbeiten. Im folgenden Kapitel wird deswegen zunächst auf die Bedürfnisse und das Vorgehen der Entwickler bei der Implementierung eingegangen. Anschließend werden nach der *MoSCoW-Priorisierung*[1] Anforderungen an das Konzept dieser Arbeit aufgestellt.

2.1. PROGRAMMVERSTEHEN

Das Programmverstehen ist ein wichtiger Prozess bei der Wartung und Erweiterung von Softwaresystemen [KT09]. Im Kontext der Bearbeitung einer Progammieraufgabe innerhalb eines Crowdsourcing-Projekts ist Programmverstehen ein essentieller Bestandteil der Aufgabe eines Crowdworkers. Dabei ist er auf die vorgegebene Struktur des Projekts, die Dokumentation und Aufgabenstellung des Task-Distributor angewiesen. Nicht immer kann eine Aufgabe herausgegeben werden, die komplett frei von Bedingtheiten zu anderen Teilen und Funktionen des Projekts ist. Bestehen Abhängigkeiten zu noch nicht existierenden Lösungen, müssen diese als *Mock-Objekte* vom Task-Distributor vorgegeben werden und der Crowdworker muss Kenntnis über die Funktionalitäten erhalten [MFC01]. Um diese Informationen erfolgreich zu vermitteln, muss zunächst analysiert werden, wie Entwickler beim Verstehen von Programmen (Softwareprojekten) vorgehen.
Die kognitiven Aspekte des Programmverstehens wurden vorwiegend in den 1980er-Jahren untersucht [KT09]. Forscher ergründeten, welche Strategien ein Entwickler beim Verstehen von Programmen verfolgt. Dabei wird grundlegend zwischen den beiden Ansätze *Top-Down* und *Bottom-Up* unterschieden. Der *Top-Down* Ansatz geht davon aus, dass die einarbeitende Person Hypothesen über die Funktionalität des Programms aufstellt, ohne den Quelltext analysiert zu haben. Bei Hypothesen handelt es sich um Annahmen über das Verhalten eines Softwaresystems. Diese Hypothesen werden auf niedrigerer Abstraktionsebene, beispielsweise auf Quelltextebene, überprüft. Ruven Brook, ein Verfechter des Top-Down-Modells, begründet diese Methode des Programmverstehens anhand des zugrundeliegenden Programmierprozesses bzw. Entstehungsprozesses des Projekts.

[1] `https://en.wikipedia.org/wiki/MoSCoW_method` (14.3.2017)

Der Programmierprozess ist für ihn die "*Konstruktion von Abbildungen von einem Anwendungsbereich über ein oder mehrere Zwischenbereiche auf einen Programmierbereich*" [KT09]. Um das erstellte Programm wiederum zu verstehen, werden die Abbildungen, welche bei der Entwicklung als Grundlage dienten, rekonstruiert. Dabei können die Abbildungen und deren Verbindungen als baumähnliche Struktur betrachtet werden, in der die Hypothese die Wurzel bildet. Diese beschreibt, welche Funktionalität der Programmierer vom Programm erwartet. Die Annahme wird so lange verfeinert und spezialisiert, bis diese präzise genug ist, um sie anhand des Quellcodes und der Dokumentation zu überprüfen. Die gesuchten Stellen im Code definiert Ruven Brook als *Beacons*. Ein Beispiel für ein *Beacon* ist ein verschachteltes Paar von Schleifen, in denen Elemente miteinander verglichen und vertauscht werden. Ein solcher Codeabschnitt würde die Aufmerksamkeit des Programmierers wecken, wenn dieser beispielsweise nach einer Sortierung von Daten innerhalb der Anwendung sucht [Bro83].
Eine weiteres kognitives Modell für Programmverstehen ist der *Bottom-Up* Ansatz. Bei diesem Verfahren folgt der Programmierer dem genau entgegengesetzten Weg. Anhand des Quelltextes erschließt sich der Entwickler nach und nach die Funktionalitäten und Zusammenhänge des Systems. Shneiderman und Mayer verfeinern das Bottom-Up Modell, indem sie zusätzlich zwischen zwei Arten von Wissen über ein Programm [SM79] unterscheiden. Zum einen das sprachabhängige *syntaktische* Wissen über grundlegende Komponenten und Anweisungen im Quelltext, zum anderen das sprachunabhängige semantische Wissen. Dieses entsteht schrittweise und lässt den Entwickler ein mentales Modell über den Anwendungsbereich bilden [And06]. Weiterhin unterscheidet man zwischen zwei Vorgehensweisen beim Programmverstehen[KT09]:

opportunistisch Der Entwickler konzentriert sich auf relevante Codezeilen, die seiner Meinung nach essentiell für die Lösung der gestellten Aufgabe sind.

systematisch Der Entwickler geht in einer festgelegten Reihenfolge vor, um das gesamte System zu verstehen, beispielsweise die zeilenweise Abhandlung des Quellcodes.

Neben den kognitiven Aspekten beschäftigt sich die Psychologie außerdem mit der Frage, welche weiteren Faktoren Einfluss auf die Verständlichkeit von Programmen haben. Dabei wurden folgende Ergebnisse recherchiert [KT09]:

- Programmeigenschaften
  - Dokumentation
  - Programmiersprache
  - Sprachen zugrundeliegendes Programmierparadigma
- individuelle Eigenschaften des Programmierers
  - Programmiererfahrung
  - Kreativität
- Aufgabenabhängigkeit
  - Aufgabenart (Wartungsaufgabe, Erweiterung)

Das Konzept dieser Arbeit muss diese Eigenschaften betrachten und deren Relevanz für die Lösung evaluieren. Weiterhin müssen die Faktoren so aufbereitet oder genutzt werden, um das Programmverstehen maximal zu fördern.

2.2. ANFORDERUNG AN SYSTEME FÜR DIE VERBESSERUNG DES PROGRAMMVERSTEHENS

Um Anforderungen an die eigene Lösung stellen zu können, werden zunächst schon erarbeitete Kriterien für Umsetzungen von Programmverständnis fördernden Tools untersucht. Es existiert dabei schon eine Vielzahl an Software Exploration Tools, die den Programmierer beim Programmverstehen unterstützen. Hausi Müller stellt in seiner Arbeit kognitive Elemente vor, die bei der Entwicklung von Software Exploration Tools beachtet werden sollen [SFM99]. Abbildung 2.1 zeigt eine Übersicht der zusammengefassten Anforderungen.

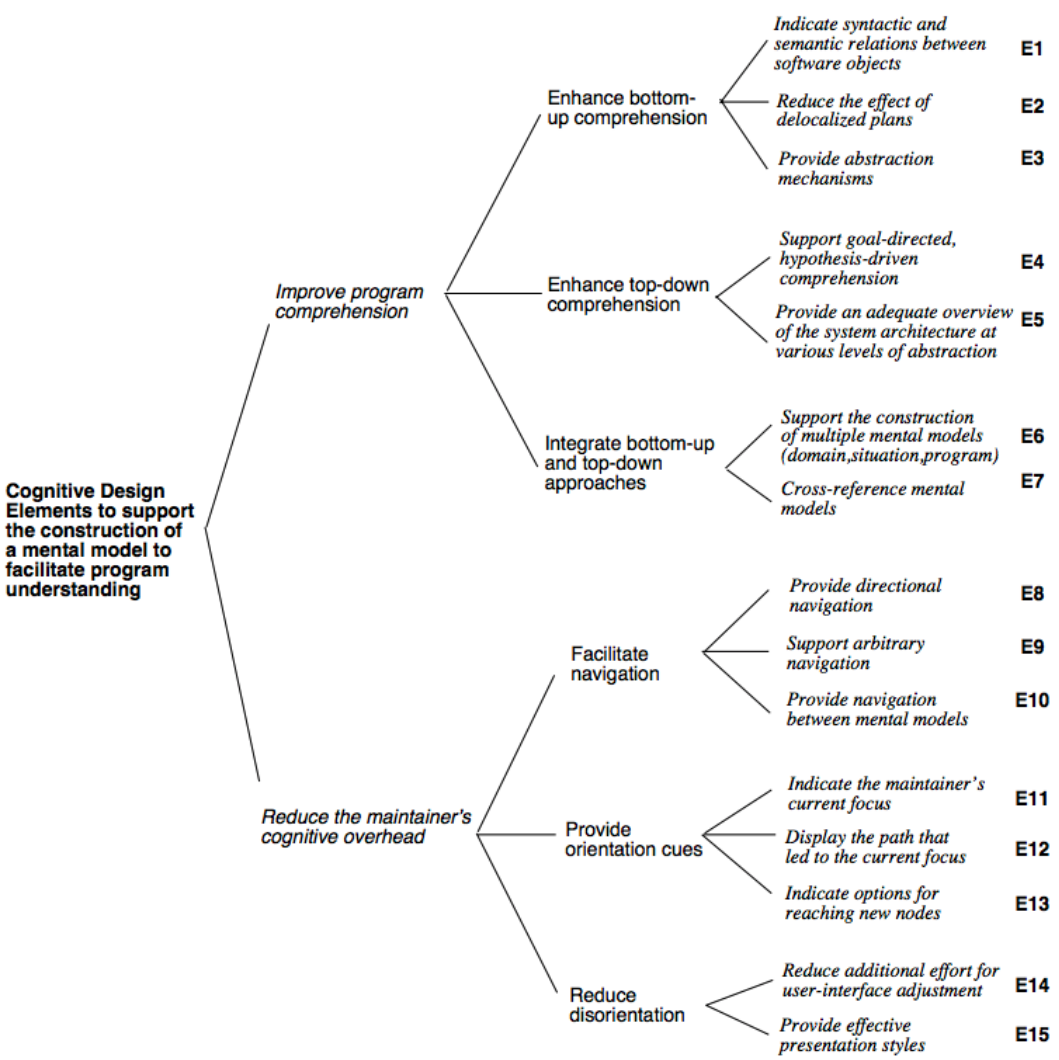


Abbildung 2.1.: Kriterien für kognitive Designelemente in Software Exploration Tools [SFM99]

Im Folgenden sollen die von Müller erarbeiteten Kriterien und deren Relevanz für diese Arbeit diskutiert werden. Zunächst findet eine Unterteilung zwischen Steigerung des Programmverständnisses und Reduzierung der kognitiven Belastung statt.

Um das Bottom-Up Vorgehen der Entwickler zu unterstützen beschreibt **E1**, dass syntaktische und semantische Beziehungen zwischen Softwareobjekten dargestellt werden sollen. Müller beschreibt in diesem Punkt eine grafische Darstellung der Beziehungen und nennt Beispiele von Tools, die diese Visualisierung ermöglichen. **E2** weist auf das Problem hin, dass die Aufteilung von Programmlogik zu einer Funktionalität des Systems über mehrere Dateien hinweg zu einem erhöhten Einarbeitungsaufwand führt. Als Lösung wird *Program Slicing* genannt, das durch eine statische Analyse des Quellcodes ein Projekt in Komponenten aufspaltet, welche jeweils eine Funktion des Systems beschreiben [BE94]. **E3** ist für diese Arbeit zunächst weniger relevant. Es wird beschrieben, dass Entwickler beim Prozess der Abbildung von Quellcode zum mentalen Modell (Bottom-Up) die Möglichkeit erhalten sollen, ihre Erkenntnisse digital festzuhalten.
Um den Top-Down Prozess zu unterstützen, nennt Müller in **E4** und **E5** ungenaue Anforderungen an das System. Hauptsächlich beschreibt er, dass Entwickler schrittweise durch den Top-Down Prozess geführt werden sollen. Das *TLES System* (Tool for Layered Explanation of Software), welches Müller im Zusammenhang nennt, dient eher zur Anreicherung der Dokumentation, als zur Hilfe des Programmverstehens und ist für diese Arbeit zunächst irrelevant.
In **E6** und **E7** fordert Müller, dass Tools unterschiedliche visuelle Darstellungen von mentalen Modellen ermöglichen sollen. Dabei nennt er Tools und beschreibt deren Visualisierungen. So werden Funktions- und Variablenaufrufe beispielsweise als grafische Darstellung präsentiert oder textuelle Repräsentationen von Quellcode mit *Program Slices* farblich aufbereitet [SFM99]. Da Nutzer zwischen den verschiedenen Modellen hin und her springen, sollen zusätzlich Referenzen zwischen den unterschiedlichen Darstellungen hervorgehoben werden.
Die untere Hälfte von Abbildung 2.1 beinhaltet Anforderungen, um die kognitive Belastung des Entwicklers zu minimieren. Dabei werden hauptsächlich Kriterien benannt, welche für die visuelle Darstellung von großen Informationsmengen entscheidend sind. Die Elemente **E8**, **E9** und **E10** stellen zunächst Anforderungen an die Navigation durch Softwareprojekte. **E8** fordert eine gerichtete und sequentielle Navigation durch den Quellcode. Als positives Beispiel nennt Müller Imagix[2], welches dem Nutzer unter anderem eine HTML Ansicht des Quellcodes im Browser ermöglicht. Neben der geführten Navigation fordert **E9** ebenfalls die Möglichkeit einer willkürlichen Vorgehensweise. Für die Lösung dieser Arbeit wäre eine solche Funktionalität zunächst nicht förderlich, da die Gefahr besteht, dass Crowdworker sich in unwichtigen Abschnitten verlieren. Zusätzlich besteht für den Task-Distributor die Gefahr, dass der Crowdworker Zugang zu Teilen des Projektes erhält, welche nur intern einsehbar sein sollten.
Element **E10** erweitert die Forderung aus **E7**. So soll durch die in **E7** geforderten Referenzen eine Navigationsmöglichkeit zwischen den verschiedenen Darstellungen der mentalen Modelle existieren. Gleichzeitig benennt Müller, dass diese Forderung zu einer großen Menge an *one-to-many* oder *many-to-one* Verlinkungen führen kann. Aufgrund dessen sollte die Anzahl verschiedener Darstellungen in der Lösung dieser Arbeit möglichst gering gehalten werden.
Die folgenden drei Anforderungen beziehen sich auf das essentielle Thema dieser Arbeit, den Entwickler bei der Orientierung innerhalb des Projekts zu unterstützen und ihm zu verdeutlichen, in welchem Bereich der Software er sich befindet. So kann es zu einer erhöhten kognitiven Belastung führen, wenn Nutzer neben Quellcode gleichzeitig Diagramme mit Beschreibungen der Funktionalitäten geboten werden. In **E11** wird empfohlen, fokussierte Elemente in allen vorliegenden Darstellungen parallel erkenntlich zu machen. Da in der textuellen Ansicht eines Projekts nicht immer alle relevanten Codeabschnitte auf einen Blick präsentiert

[2] `https://www.imagix.com` (13.4.2017)

werden können, müssen zusätzliche Informationen direkt in abstrakteren Abbildungen hervorgehoben sein. Die Frage, wie relevante Abschnitte gefunden werden können, klärt Müller nicht. Dafür nennt er Möglichkeiten, wie die Stärke der Relevanz visuell aufbereitet werden kann. Eine für diese Arbeit vorstellbare Lösung wäre die unterschiedliche Größendarstellung einzelner Komponenten oder die Verwendung der *Fish-Eye* Technik [LA94]. **E12** erweitert den Gedanken aus **E11** und fordert gleichzeitig eine Darstellung, wie der Nutzer bis zum aktuellen Abschnitt vorgegangen ist. In der Lösung dieser Arbeit könnten Codestellen hervorgehoben werden, die der Task-Distributor beim Erstellen der Tests aufgerufen hat. In Element **E13** wird gefordert, dass Verlinkungen zu weiteren wichtigen Erläuterungen eines Elements hervorgehoben werden sollen. Als Beispiel nennt er Hyperlinks in HTML-Dokumenten, die durch eine bestimmte Formatierung für den Nutzer relevant erscheinen.
Da bei der Lösung dieser Arbeit wenige Darstellungsformen geplant sind, ist **E14** zunächst nicht relevant. **E15** beschreibt dagegen den sehr wichtigen Fakt, einen effektiven Präsentationsstil zu verwenden. Dabei nennt Müller leider nur Beispiele von Tools, welche angepasste Layoutalgorithmen für Graphendarstellungen einführen, um die Komplexität der Abbildung für den Betrachter zu senken.
Ein Großteil der von Müller aufgestellten Kriterien sollten auch in dieser Arbeit für die Gestaltung des Interfaces beachtet werden. Dabei sind Forderungen nach Unterstützung von vielen mentalen Modellen zunächst weniger relevant für die zu erarbeitende Lösung. Um das Top-Down und Bottom-Up Vorgehen zu unterstützen wird zunächst je eine Darstellungsform umgesetzt. Diese Beschränkung auf wenige Darstellungsformen minimiert den Navigationsaufwand für die Nutzer. Wichtiger sind zunächst Kriterien für die Minimierung der kognitiven Belastung und der Navigation innerhalb des Softwareprojekts.

2.3. ANFORDERUNGEN AN DIE LÖSUNG DIESER ARBEIT

Aus den vorangegangenen Abschnitten lassen sich nun Anforderungen an die Lösung dieser Arbeit stellen. Neben den Kriterien, um den Entwickler bei der Bearbeitung einer Aufgabe innerhalb eines Softwareprojektes zu unterstützen, sollte die Lösung ebenfalls in das zugrundeliegende Konzept einer *testgetriebenen Crowdsourcing Plattform für Softwareentwicklung* (Kapitel 4.1) integriert werden können. Aus diesem Grund kommen, zusätzlich zur Verbesserung des Programmverstehens, kontextbedingte Anforderungen hinzu. Zum einen durch das Thema Outsourcing von Softwareentwicklung in Form von Crowdsourcing, zum anderen durch den testgetriebenen Ansatz des Plattform-Konzepts.
Um die Bereiche auszuarbeiten, in denen der Programmierer unterstützt werden muss, wird der auf der Plattform zugrundeliegende Ablauf mit Hilfe einer *Hierarchical Task Analysis* (HTA) in Teilaufgaben dargestellt (siehe Abbildung 2.2). Dabei können die zu erfüllenden Aufgaben in drei Bereiche gegliedert werden.
Wie in der Softwareentwicklung üblich, beginnt ein Entwickler mit der **Analyse** der Aufgabe. Für die Qualität der Lösung ist eine klare Aufgabenstellung sowie eine Kontextbeschreibung wichtig. Hat er sich einen Überblick über das vorliegende Projekt gemacht, setzt er mit der **Implementierung** der Lösung fort. Durch die auf der Crowdsourcing-Plattform vorgegebenen Softwaretests erhält der Entwickler fortlaufend **Feedback** zu seinem Fortschritt. Werden die festgelegten Anforderungsspezifikationen nicht erfüllt, wechselt der Entwickler meist wieder in die **Analyse**. Der Lösungsprozess ist somit iterativ und sollte vom Konzept dieser Arbeit entsprechend unterstützt werden.

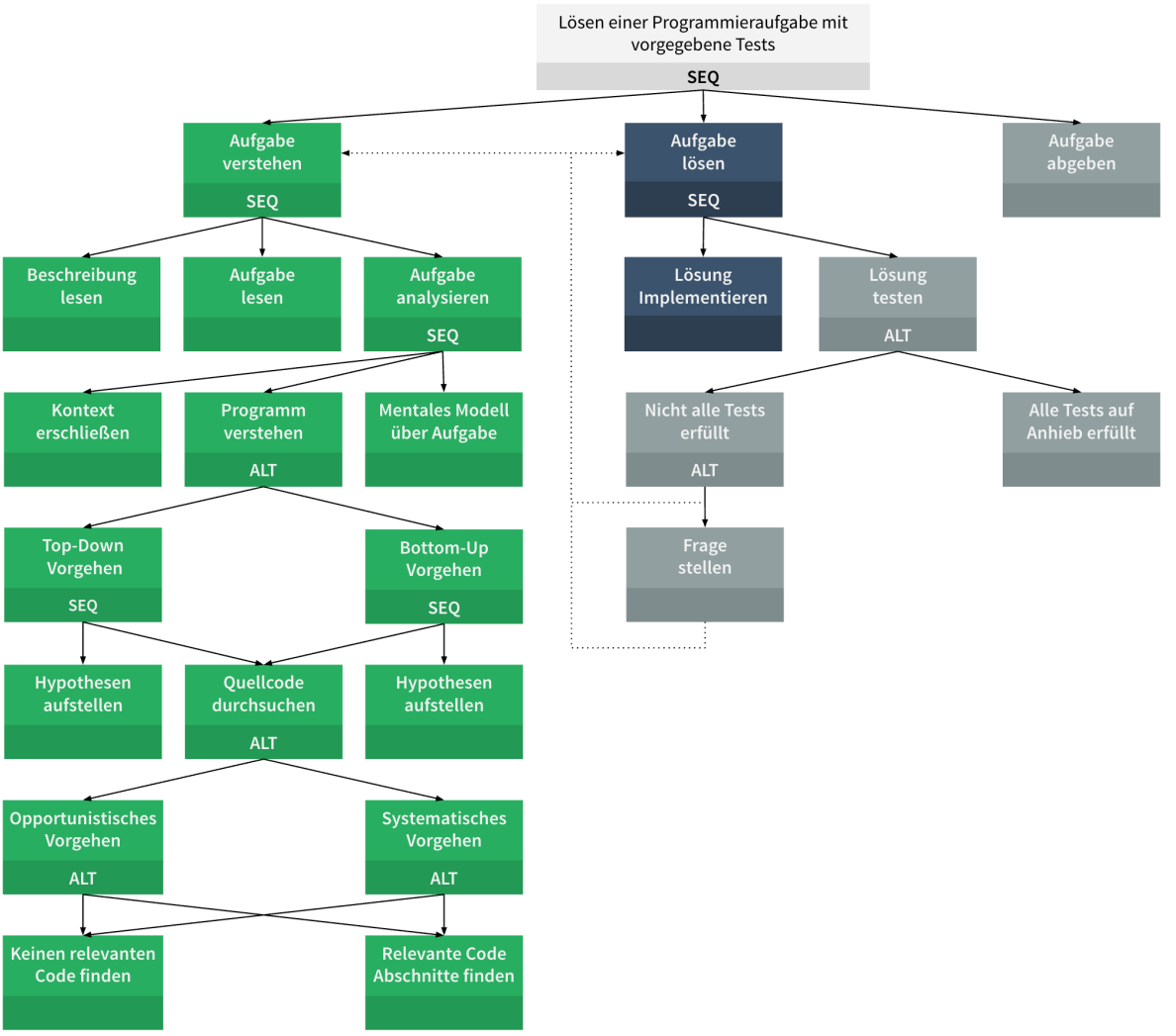


Abbildung 2.2.: HTA (Hierarchical Task Analysis) für das Lösen einer Aufgabe mit vorgegebenen Tests. Sequenzen, die mehrmals auftreten können, werden in diesem Diagramm durch eine gestrichelte Linie gekennzeichnet, was von der üblichen HTA-Definition abweicht. Farbliche Kodierung: Grün - Analyse, Blau - Implementierung, Grau - Feedback

Abbildung 2.3 gibt zunächst einen groben Überblick über die aufgestellten Anforderungen für eine Webanwendung, welche die Bearbeitung kleiner Programmieraufgaben ermöglicht. Diese erlauben später eine Evaluierung des erarbeiteten Konzepts. Dabei werden die Anforderungen in die drei Phasen (**Analyse** - **Implementierung** - **Feedback**), welche durch die HTA extrahiert wurden, eingeteilt.
Zunächst muss die **Analyse**-Phase des Entwicklers unterstützt werden. Hierfür wurde in den vorhergehenden Kapiteln der Fokus auf das Programmverstehen und schon erarbeitete Anforderungen gelegt. Die recherchierten Kriterien sind dabei zum Großteil übernommen worden. Zusätzlich muss die Aufgabenbeschreibung klar definiert werden, um den Crowdworker in die Aufgabe einzuführen.

Bei der eigentlichen Umsetzung der Aufgabe muss das Konzept den Nutzer bei der **Implementierung** unterstützen. Hierfür sollten, die in Editoren etablierte Funktionalitäten wie *Codecompletion* und *Hover-Tooltips*, umgesetzt werden. Gleichzeitig sollen diese mit aufgabenspezifischen Informationen angereichert werden können, um dem Nutzer weiteres Wissen zu vermitteln und so das Programmverstehen zu erleichtern. Um die kognitive Belastung und den Navigationsaufwand für Crowdworker zu senken, muss das Projekt auf die für die Lösung relevanten Stellen eingeschränkt werden.
In der **Feedback**-Phase sollte der Crowdworker die Möglichkeit erhalten, direkten Kontakt zum Task-Distributor aufnehmen zu können. Um den Kommunikationsaufwand zwischen beiden Parteien zu senken, müssen Probleme klar definiert werden können. Weiterhin sollen die vom Task-Distributor erarbeiteten Tests auf der Plattform ausführbar sein. Die Fehlerausgabe der Tests sollte Crowdworkern ein verständliches Feedback liefern, um den iterativen Entwicklungsprozess nicht zu unterbrechen.

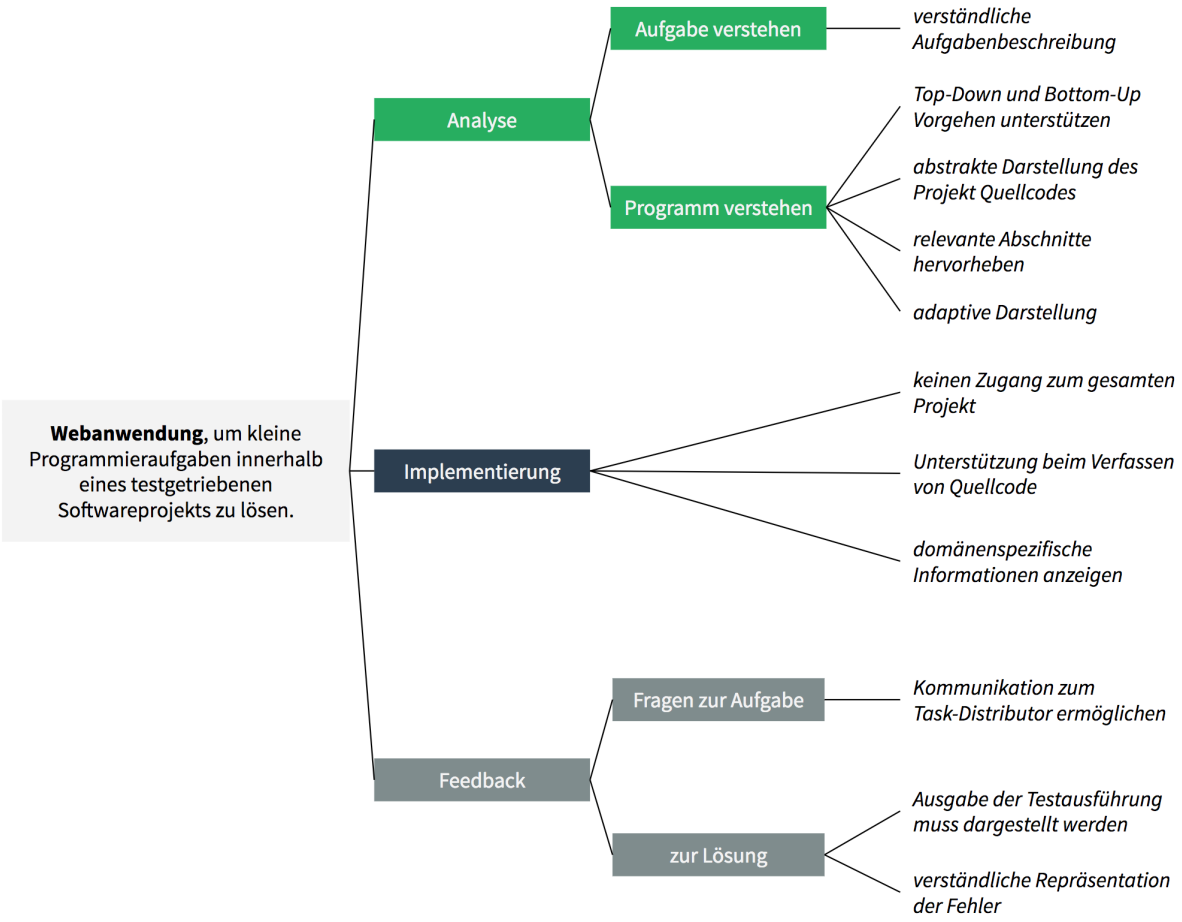


Abbildung 2.3.: Anforderungen an das Konzept dieser Arbeit

Tabelle 2.1 listet die Anforderungen für das zu entwickelnde Webinterface nach der MoSCoW-Priorisierung auf. Dabei wird zusätzlich zu den Priorisierungsstufen zwischen den drei Phasen Analyse, Implementierung und Feedback unterschieden. Da die Anforderungen zum einen aus der Analyse des Programmverstehens, zum anderen aus dem Kontext einer testgetriebenen Crowdsourcingplattform für Softwareentwicklung entstammen, wird eine dritte Spalte der Tabelle hinzugefügt, welche die Grundlage des Kriteriums kennzeichnet (**TDD-P** - testgetriebene Crowdsourcingplattform, **PV** - Programmverstehen).
Die *MUST*-Kriterien müssen erfüllt sein, damit Crowdworker in der Lage sind Aufgaben annehmen und bearbeiten zu können. Dabei ist es essentiell, dass die erarbeitete Lösung Programmiersprachen unabhängig funktioniert, um verschiedenste Softwareprojekte über die Crowdsourcingplattform entwickeln zu können. Deshalb werden zunächst symbolisch drei Programmiersprachen (Python, PHP und Javascript) als Überprüfung für die Sprachenunabhängigkeit des erarbeiteten Konzepts gewählt. Die *MUST*-Kriterien bilden somit vor allem die technischen Anforderungen. Die weiteren Kriterien sind notwendig, um den Lösungsprozess des Entwicklers (Abbildung 2.2) zu unterstützen und die Qualität des Konzepts dieser Arbeit evaluieren zu können.

MUST		
Analyse		
M10	Aufgabentitel und Aufgabenbeschreibung müssen verständlich dargestellt werden	TDD-P
Implementierung		
M20	Verfassen von Quellcode muss möglich sein	TDD-P
M30	Nur für den Nutzer freigegebene Dateien dürfen zugänglich sein	PV
Feedback		
M40	Implementierter Quellcode muss ausgeführt bzw. getestet werden können	TDD-P
M50	Ausgabe der Testausführung muss textuell dargestellt werden	TDD-P
SHOULD		
Analyse		
S10	Programmverstehen für den Crowdworker erleichtern (vgl. **E1** - **E7** Abbildung 2.1) • Top-Down und Bottom-Up Vorgehen unterstützen • abstrakte Darstellung des Quellcodes (mindestens eine Stufe höher als textueller Quellcode) • relevante Abschnitte hervorheben • kognitive Belastung des Crowdworker mindern (vgl. **E8** - **E15** Abbildung 2.1)	PV
Implementierung		
S20	Nutzer bei der Implementierung unterstützen • domänenspezifische Informationen zur Aufgabe müssen eingepflegt und visualisiert werden können • kontextbasierte Codevervollständigung • kontextbasierte Hover Hilfe • Syntaxhervorhebungen (Python, PHP und Javascript)	PV

Feedback S30	Fehlerausgaben müssen verständlich dargestellt werden	TDD-P
COULD		
Analyse C10	Verschiedene Abstraktionsebenen des Quellcodes visualisieren	PV
Implementierung C20	Refactoring, automatische Formatierung und GoTo-Funktionalität für Python, PHP und Javascript	TDD-P
C30	Sinnvolle Verlinkung zwischen Code-Implementierung und Code-Exploration (vgl. E11 Abbildung 2.1)	PV
Feedback C40	Kommunikationskomponente, um Fehler zu melden oder Fragen zu stellen	TDD-P
WON'T		
Analyse W10	Adaptive Darstellung nach Bedürfnissen des Nutzers und angenommener Aufgabe	PV
Implementierung W20	Adaptierung der Benutzeroberfläche durch den Nutzer (visuelle Darstellung und Interaktionsmöglichkeiten)	TDD-P

Tabelle 2.1.: Anforderungen an die Lösung dieser Arbeit

3. STATE OF THE ART / VERWANDTE ARBEITEN

Das folgende Kapitel stellt existierende Web-Entwicklungsumgebungen vor, um die technische Grundlage für die Umsetzung dieser Arbeit zu schaffen. Zunächst wird das *Language Server Protocol* beschrieben, welches in verwandten Arbeiten Verwendung findet. Weiterhin werden Konzepte vorgestellt, welche Crowdsourcing für Softwareentwicklung ermöglichen. Dabei wird der Fokus auf deren Evaluation gelegt, um bestehende Probleme zu erfassen und im eigenen Konzept zu lösen oder zu umgehen.

3.1. LANGUAGE SERVER PROTOCOL

Das *Language Server Protocol*[1] ist ein von Microsoft[2] bereitgestelltes Protokoll, für die Kommunikation zwischen einer Softwareanwendung (Client) und einem *Language-Server*, um dem Client Funktionalitäten wie sprachabhängige Autovervollständigung, Codenavigation und automatische Codeformatierung zu ermöglichen. Grund für die Entwicklung eines solchen Projekts war es, eine einheitliche Lösung für Editoren zu schaffen, um sämtliche Programmiersprachen Unterstützungen von Grund auf anbieten zu können. So sollen Entwickler nicht Editor spezifische Plugins implementieren, sondern eine Lösung umsetzen, welche von allen Tools genutzt werden kann. Hierfür sollen Entwickler für jede Sprache einen zugehörigen *Language-Server* programmieren, welcher über das *Language Server Protocol* angesteuert werden kann. Um das Projekt voranzutreiben, haben Firmen wie Codenvy[3], Red Hat[4] und Sourcegraph[5] ihre Unterstützung zugesichert.
Aufbau des Protokolls mit Header und Content Teil ist vergleichbar mit dem *Hypertext Transfer Protocol* (HTTP). Der Header besteht aus den Feldern *Content-Length* und *Content-Type*. Der Content Abschnitt beinhaltet die eigentliche Anfrage an den Server unter der Verwendung des *JSON RPC Protokolls* [JSO10].

[1] https://github.com/Microsoft/language-server-protocol (25.3.2017)
[2] `https://www.microsoft.com/de-de/` (8.7.2017)
[3] https://codenvy.com (25.3.2017)
[4] http://redhat.com (25.3.2017)
[5] https://sourcegraph.com (25.3.2017)

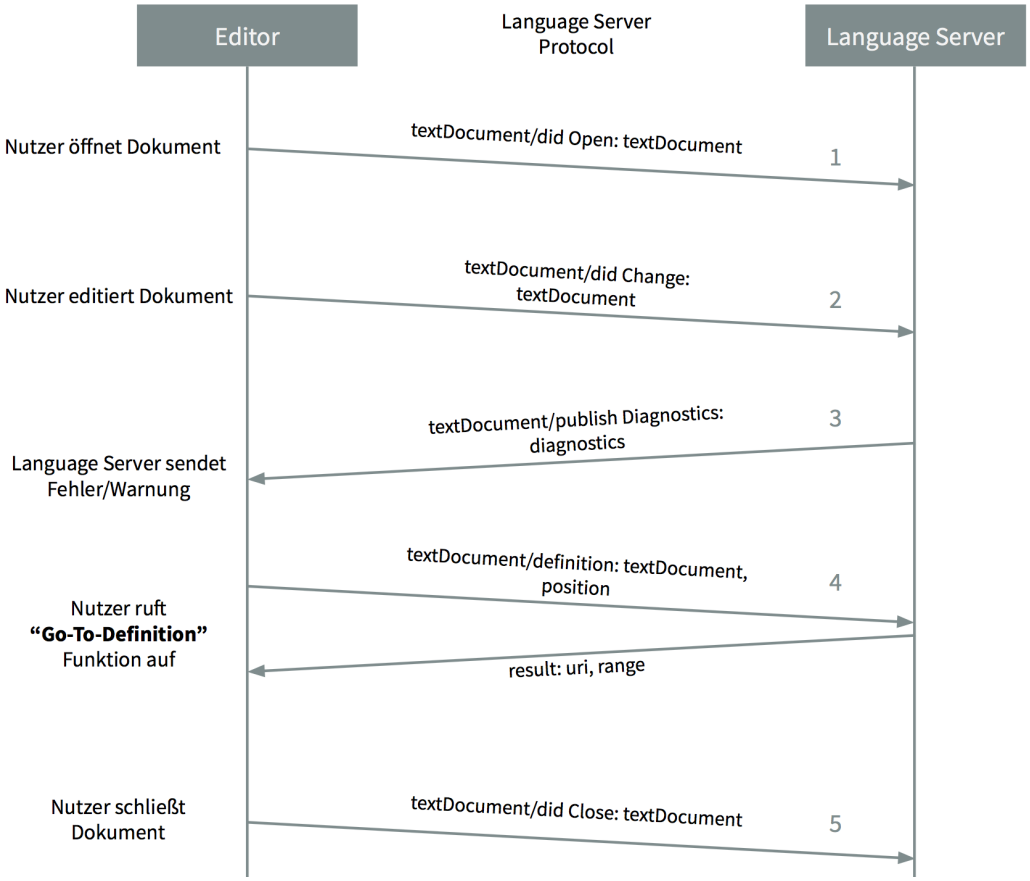


Abbildung 3.1.: Kommunikation zwischen Tool und Language-Server (`https://github.com/Microsoft/language-server-protocol` (21.8.2017) angepasst)

Abbildung 3.1 veranschaulicht anhand eines Beispiels, wie die Kommunikation zwischen Client und Server abläuft. Zunächst öffnet der Nutzer eine Datei im Editor. Parallel wird ein Request an den Language-Server gesendet, um diesem Zugang zur geöffneten Datei zu ermöglichen (**1**). Die Anfrage (Request) enthält dafür den Inhalt und den Pfad zur geöffnetet Datei in Form eines *Textdocument* Elements. Die Struktur bzw. das Interface eines solchen Elements wird in der *Language Server Protocol* Spezifikation definiert[6]. Daraufhin editiert der Nutzer die Datei und das Tool teilt dem Language-Server die Veränderung mit (**2**). Dieser analysiert den empfangenen Quellcode. Da der Nutzer einen Syntaxfehler verursacht hat, sendet der Language-Server eine Fehler-Response (**3**). Diese beinhaltet eine Liste aus *Diagnostic* Elementen, welche vom Editor verarbeitet werden müssen. Im nächsten Schritt führt der Nutzer im Editor einen GoTo-Befehl aus, mit dem er zu einer bestimmten Definition im Quellcode springen möchte. Der Editor sendet eine Anfrage mit dem Pfad des Dokuments und der Position innerhalb des Editors und erhält den Dateipfad(uri) und die Stelle (Range) der angeforderten Methode oder Variable (**4**). Die letzte Aktion des Nutzers schließt die Datei und der Editor teilt dem Language Server diese Aktion mit (**5**).

[6] `https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md` (17.7.2017)

Derzeit existieren schon einige Umsetzungen von Language-Servern für verschiedene Sprachen[7]. Standardmäßig verwendet werden diese in Visual Studio Code[8], Codenvy und Eclipse Che[9] (Stand 8.7.2017). Neue Projekte, wie die Theia IDE[10], die auf das Language-Server-Protokoll setzen und Language-Server Implementierungen von Firmen wie der Eclipse Foundation[11], zeigen den positiven Zuspruch für die Idee eines einheitlichen Sprachprotokolls.
Für das Konzept dieser Arbeit ist die Verwendung des Language-Server-Protokolls eine sinnvolle Entscheidung. So muss bei der Wahl des Editors nicht auf eine vorhandene Sprachunterstützung geachtet werden, da diese Funktionalitäten von den Language-Servern bereitgestellt sind.

3.2. WEBBASIERTE ENTWICKLUNGSUMGEBUNGEN

Immer mehr Desktop-Applikationen werden zu Webanwendungen portiert. Auch Entwicklungsumgebungen erhalten Webableger, um kollaboratives Arbeiten und neue Feedbackmethoden zu forcieren [KKVV12]. Weitere Vorteile, die im Zusammenhang mit Entwicklung im Web genannt werden, sind:

- kein lokaler Installationsaufwand
- skalierbare Rechenleistung
- leistungsstarke Rendering Engines der Browser
- Zentralisierung der Konfiguration und des Deployments

Gerade bei der Umsetzung einer crowdbasierten Lösung sind diese Eigenschaften von Vorteil. Im folgenden Abschnitt werden deshalb relevante Vertreter webbasierten Entwicklungsumgebungen vorgestellt. Dabei wurde der Fokus auf Open Source Projekte gelegt, um die Verwendung im eigenen Konzept sicherzustellen.

3.2.1. ECLIPSE ORION

Eclipse Orion[12] ist ein Top-Level Open-Source Projekt, welches von der Eclipse Cloud Development Community[13] 2011 gestartet wurde. Ziel ist die Implementierung einer cloud-basierten Entwicklungsumgebung für die Umsetzung webbasierter Anwendungen. Vor der Veröffentlichung als eigenes Produkt war die IDE Teil des Firefox Browser. Die Komponenten von Eclipse Orion sind in Javascript umgesetzt. Da die Entwickler ihren eigenen Editor für Implementierung genutzt haben, ist Javascript die am stärksten unterstützte Sprache. Serverseitig wird auf REST APIs gesetzt. Die einzelnen Komponenten können in verschiedenen Kombinationen von Entwicklern verwendet und verteilt (*deployt*) werden. Abbildung 3.2 zeigt vier verschiedene Möglichkeiten für den Einsatz und die Verwendung von Eclipse Orion [Mac14].

[7] `http://langserver.org` (8.7.2017)
[8] `https://code.visualstudio.com` (7.7.2017)
[9] `https://www.eclipse.org/che/` (7.7.2017)
[10] `https://github.com/theia-ide/theia` (8.7.2017)
[11] `https://eclipse.org/org/foundation/` (23.3.2017)
[12] `https://wiki.eclipse.org/Orion` (4.4.2017)
[13] `http://www.eclipse.org/ecd/` (4.4.2017)

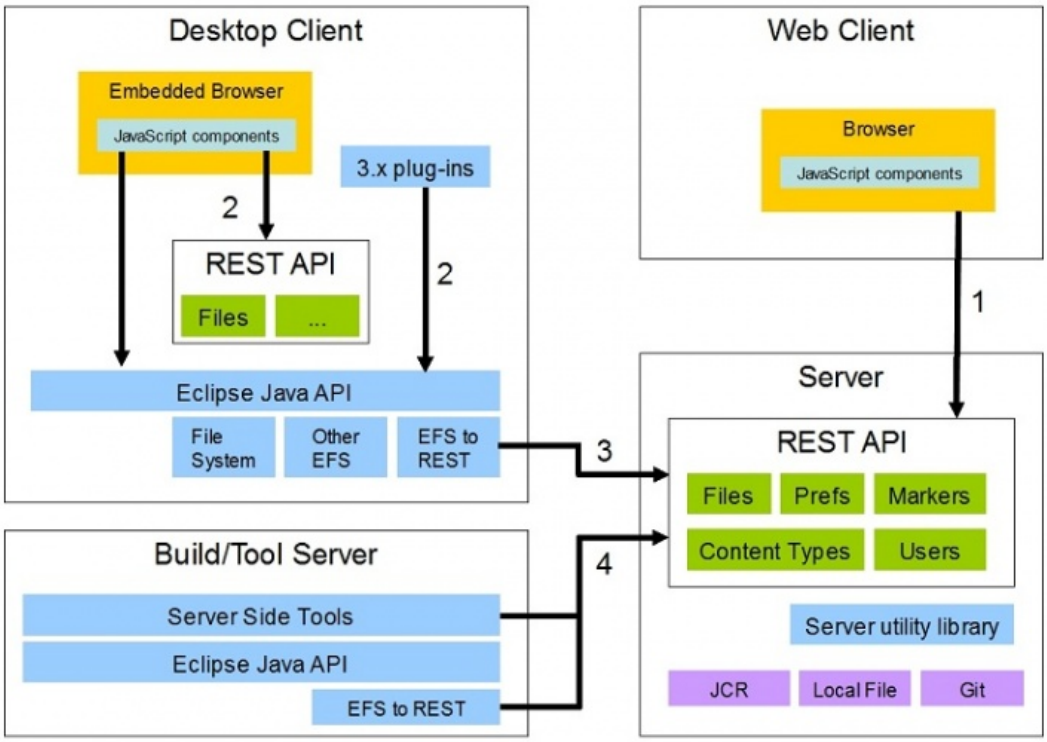


Abbildung 3.2.: Eclipse Orion Architektur und Nutzungsmöglichkeiten [Mac14]

1. **Browser Client / entfernte Daten**: Ein webbasierter Browser nutzt die Orion Javascript Client Bibliotheken, welche mit den Orion Services auf einem Server über eine REST API kommunizieren. Die Orion Server Komponente verwaltet in diesem Fall die benötigten und erstellten Ressourcen (zum Beispiel Quellcode und Nutzerprofile).
2. **Mixed Client / lokale Daten**: Ein Rich Client (in diesem Beispiel ein Eclipse Client) beinhaltet Orion Javascript Bibliotheken und REST APIs. Um diesen an den Rich Client zu koppeln, werden Plugins geschrieben, die mit der Eclipse Java API kommunizieren. Die Daten werden dabei nicht auf einem Server abgelegt.
3. **Rich Client / entfernte Daten**: Ein klassischer *Eclipse Client* kommuniziert mit einem Orion Server über REST API, welcher die Daten auf einem entfernten Server verwaltet. Hierbei werden keine Funktionalitäten des Eclipse Orion Editors genutzt.
4. **Remote Client / entfernte Daten**: Serverseitige Tools verwenden die Inhalte von einem Orion Workspace Server. Beispielsweise ein *Build Server* der auf Projekte des Servers zugreift.

Die Darstellung der verschiedenen Einsatzmöglichkeiten lieferte Ideen für die weitere Umsetzung des Konzepts dieser Arbeit. Durch den starken Fokus auf Javascript und der fehlenden Virtualisierungsunterstützung ist Eclipse Orion als Gesamtkonzept für eine weitere Verwendung nicht geeignet. So wäre es beispielsweise nicht möglich, jegliche Testimplementierungen ausführen zu lassen (Anforderung M40). Allerdings bietet Eclipse Orion seit Version 9 seinen integrierten Code Editor als eigenständige Komponente für Entwickler an. Diese kann in eigene Projekte eingebunden und erweitert werden. Im folgenden Abschnitt wird diese Komponente deshalb näher vorgestellt.

ORION CODE EDIT WIDGET

Orion Code Edit Widget[14] ermöglicht es Entwicklern, den in Eclipse Orion verwendeten Editor in anderen Projekten einbetten und anpassen zu können. Das Widget beinhaltet alle wichtigen Funktionalitäten der Hauptanwendung, inklusive Web Entwicklungstools wie *Tern*[15] und dem *Ancorn*[16] Parser. Diese Bibliotheken dienen der Analyse und Arbeit mit Javascript Quellcode. Die Sprachunterstützungen können je nach Bedürfnis geladen und verwendet werden, um die Ladezeiten zu verringern. Dabei bietet das Widget für Javascript die stärkste Sprachunterstützung. Für Syntaxhervorhebungen nutzt Orion Code Edit *Media Types* (MIME Types) und unterstützt, die in den Anforderungen spezifizierten Sprachen. Durch seine OSGi ähnliche *Service Registry*[17] sind Erweiterung und Anpassung des Orion Code Edit Widget einfach umzusetzen. Weiterhin zeigt das Eclipse Che Projekt, dass der Orion Editor das Language-Server-Protokoll implementieren kann.

3.2.2. ECLIPSE CHE

Eclipse Che ist ein Workspace-Server mit integrierter Entwicklungsumgebung (IDE). Entwickelt wird das Projekt im Rahmen der Eclipse Foundation und ist quelloffen unter der Eclipse Public License[18]. Das Projekt wird mit dem Google Web Toolkit (*GWT*)[19] umgesetzt und ist eine Java-Applikation. Die Grundidee ist das zentrale Entwickeln und Ausführen von Softwareprojekten innerhalb des Browsers ohne weitere lokale Installationen. Für die Nutzung von Eclipse Che, wird zunächst eine Che Instanz lokal oder auf einem Server installiert. Diese kann von mehreren Clients konfiguriert und verwendet werden. Nutzer können dabei *Che Workspaces* anlegen (Abbildung 3.3), welche aus der Che-Entwicklungsumgebung (IDE) und einer beliebigen Anzahl von Projekten und Laufzeitumgebungen für die Ausführung bestehen. Letzteres sind Docker-Container[21]. Somit werden Projekte auf dem Server immer innerhalb von Containern ausgeführt. Diese Laufzeitumgebungen können individuell an die Anforderungen des Projektes mit Hilfe von Dockerfiles konfiguriert werden. Dabei existiert die Möglichkeit, Container-Vorlagen (*Stacks*) zu verwenden. Standardmäßig ist ein Workspace transient, über eine *Snapshot* Funktion kann der Zustand aber persistiert werden.

[14] `https://wiki.eclipse.org/Orion/How_Tos/Code_Edit` (21.4.2017)
[15] `http://ternjs.net` (24.4.2017)
[16] `https://github.com/ternjs/acorn` (24.4.2017)
[17] `https://wiki.eclipse.org/Orion/Documentation/Developer_Guide/Architecture#Object_References` (9.7.2017)
[17] `https://www.eclipse.org/che/images/hero-home-technology.png` (25.3.2017)
[18] `https://www.eclipse.org/legal/epl-v10.html` (23.3.2017)
[19] `http://www.gwtproject.org` (23.4.2017)
[21] `https://www.docker.com` (23.4.2017)

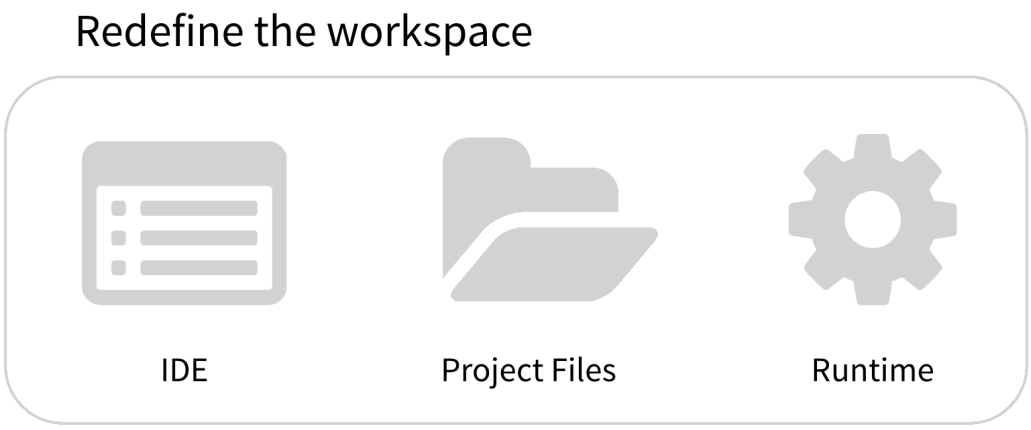


Abbildung 3.3.: Definition eines Workspaces in Eclipse Che (Angepasste Grafik[20])

Abbildung 3.4 bildet die zugrunde liegende Technologie von Eclipse Che ab. Die integrierte Che-Entwicklungsumgebung ist über den Browser erreichbar. Jegliche Änderungen und Eingaben in der Browser-IDE werden über eine REST-Schnittstelle an die Workspaces weitergeleitet. Der Workspace-Master (Abbildung 3.3 **WS Master**) verwaltet sämtliche **Workspaces**. Zusätzlich beinhalten diese eine **WS Agent**-API, um direkte Änderungen in den Umgebungen zu ermöglichen. Diese Schnittstellen bieten Entwicklern die Möglichkeit, Eclipse Che ohne die IDE als Workspace Server nutzen zu können. Das bereitgestellte *Software Development Kit* (**SDK**) ermöglicht es, die drei Komponenten (Che Server, Workspace und IDE) von Eclipse Che mit Hilfe von Plugins zu erweitern.

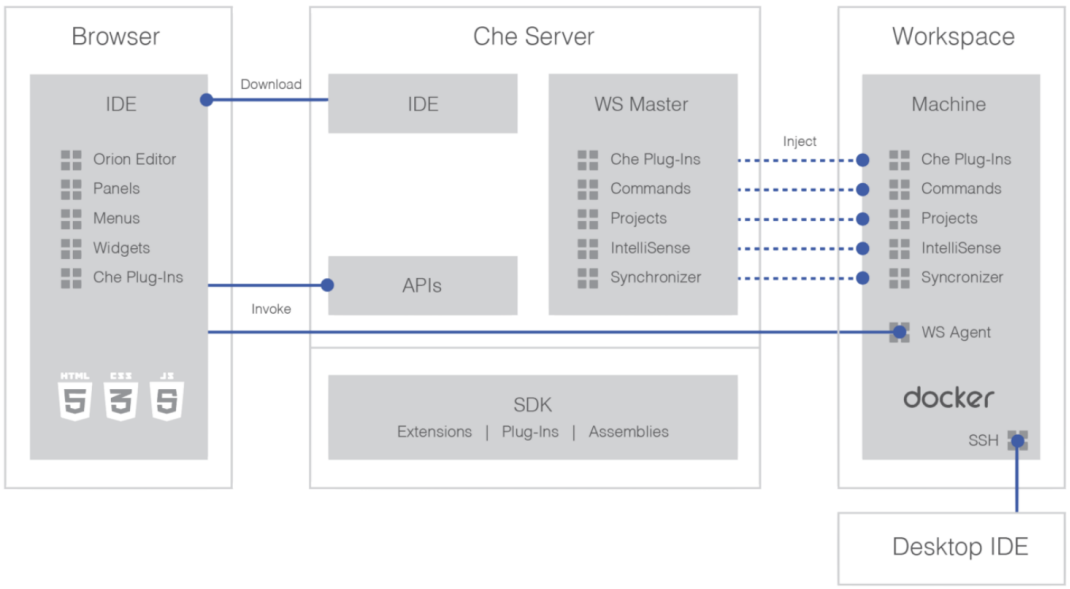


Abbildung 3.4.: Technologie Grafik von Eclipse Che (Bildquelle angepasst[22])

[22] `https://www.eclipse.org/che/docs/assets/imgs/Docker--CodenvyMeeting.png` (25.3.2017)

Der eingebettete Code-Editor (Abbildung 3.4 Orion Editor) der Entwicklungsumgebung wurde aus dem *Eclipse Orion*[23] Projekt (siehe Kapitel 3.2.1) übernommen. Dabei handelt es sich um eine auf dem Client ausführbare Javascript Anwendung. Diese kommuniziert mit den APIs des Eclipse Che Servers. Sämtliche Funktionalitäten des Editors wie Coderefaktorisierung, automatische Formatierung und Codevervollständigung werden über den Einsatz des *Language Server Protocols* (Kapitel 3.1) ermöglicht. Hierfür werden, je nach Konfiguration des Benutzers, Language-Server innerhalb eines Workspaces gestartet und eine Kommunikation über APIs bereitgestellt. Ein weiteres Feature von Eclipse Che ist die Bearbeitung eines Workspaces über lokal installierte Desktop-Entwicklungsumgebungen mittels Secure Shell (*SSH*). Hierfür stellt jeder Workspace einen SSH-Agent bereit.
Durch seinen Aufbau bietet Eclipse Che unterschiedliche Anwendungsmöglichkeiten. Die Hauptintention der Entwickler ist, die komplette Entwickung in die Cloud zu bringen. Das Gesamtkonzept ist stimmig, aber im Vergleich zu Desktop Entwicklungsumgebungen ist die Performance noch nicht auf einem vergleichbaren Niveau. Auch die hohe Anzahl an offenen GitHub Issues[24] und veröffentlichte Reviews[25] zeigen, dass die Browser-IDE noch nicht komplett ausgereift ist. Großer Vorteil zu ähnlichen Cloud-IDEs wie *Cloud9*[26], *Codeanywhere*[27] und *Nitrous*[28] ist, dass der Quellcode von Eclipse Che *Open Source* ist und somit die Möglichkeit besteht, eigene Erweiterungen einzupflegen. Hierfür existieren Dokumentationen zu den verschiedenen Komponenten von Che[29]. Weiterhin ermöglicht Che durch seine vielen APIs eine Remote-Nutzung, was es für eine weitere Verwendung in dieser Arbeit interessant macht.

3.3. KONZEPTE FÜR CROWD-BASIERTE SOFTWAREENTWICKLUNG

Neben der Technologieanalyse von Browser-Entwicklungsumgebungen ist eine Analyse von verwandten Konzepten Crowd-basierter Softwareentwicklung für diese Arbeit essentiell. Dabei sind vor allem die Aufbereitung der Aufgaben, als auch durchgeführte Evaluationen interessant, um Probleme der Nutzer zu verstehen.

3.3.1. COLLABODE

Collabode[30] ist eine webbasierte Entwicklungsumgebung von Max Goldman, basierend auf Java [Gol12]. Hauptziel des Projekts ist das *kollaborative Programmieren*. Abbildung 3.5 zeigt den Aufbau des User-Interfaces der Anwendung. Im abgebildeten Beispiel bearbeiten drei Personen gleichzeitig die dargestellte Datei. Probleme, wie die Unterbrechung der eigenen Bearbeitung durch Fehler von anderen Entwicklern, löst Collabode durch die Einführung von *Error-Mediated Integration*. Da das kollaborative Arbeiten im Kontext dieser Arbeit zunächst nicht betrachtet werden soll, wird für eine genauere Beschreibung des Verfahrens auf die Arbeit von Goldman verwiesen [Gol12].

[23] `https://orionhub.org` (23.4.2017)
[24] `https://github.com/eclipse/che/issues` (8.7.2017) (438 Issues, 128 Bugs)
[25] `https://www.heise.de/developer/artikel/Eclipse-Che-die-IDE-der-Zukunft-3266413.html` (22.8.2017)
[26] `https://c9.io` (8.7.2017)
[27] `https://codeanywhere.com` (8.7.2017)
[28] `https://codeanywhere.com` (8.7.2017)
[29] `https://www.eclipse.org/che/docs/` (19.8.2017)
[30] `http://up.csail.mit.edu/collabode/` (29.4.2017)

In seiner Arbeit stellt Goldman neue Modelle für die Nutzung von kollaborativer Programmierung vor [Gol12]. Im Folgenden werden zwei für diese Arbeit relevanten Modelle vorgestellt:

1. **Test-Driven Pair Programming**: Dieses Modell kombiniert die Methoden *Pair-Programming* und *testgetriebene Entwicklung*. Während ein Teil des Paares die Tests entwickelt, muss der andere sich um die Implementierung kümmern, um die Tests zu erfüllen.

2. **Micro-Outsourcing**: In diesem Modell setzt ein Programmierer (Task-Distributor) auf die Unterstützung einer Crowd, welche kleine Änderungen im Projekt einreichen darf. Micro-Outsourcing erlaubt dem Task-Distributor, ohne Unterbrechung seine Programmierung fortzusetzen. Gleichzeitig können Crowd-Entwickler in anderen Abschnitten des Projektes Programmcode implementieren.

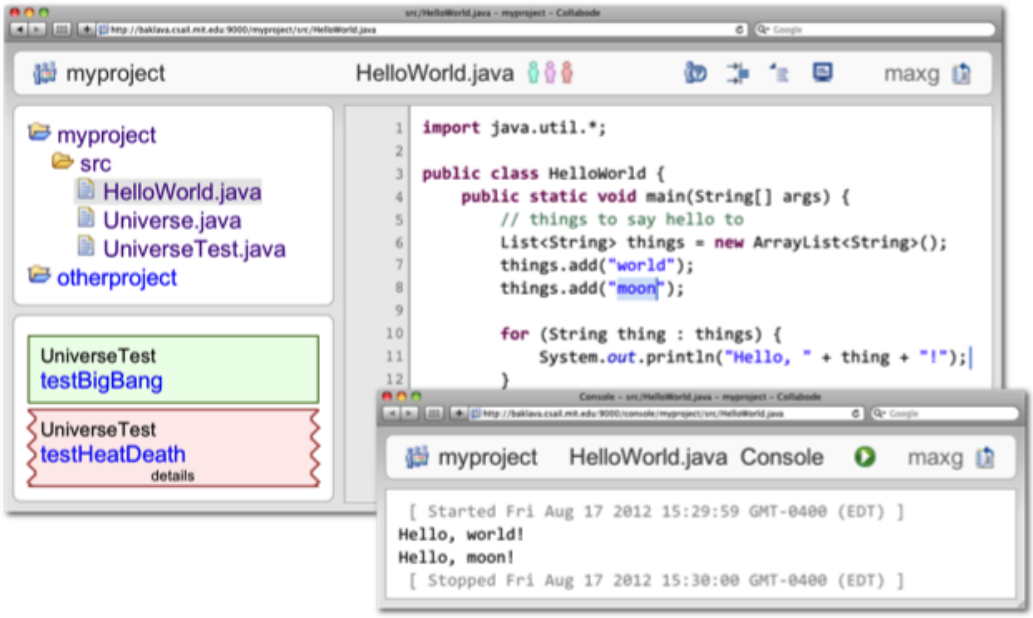


Abbildung 3.5.: Darstellung von zwei Fenstern der Collabode Entwicklungsumgebung. Das obere zeigt den Quelltext der *HelloWorld.java* Datei, die durch drei Nutzer bearbeitet wird. Auf der linken Seite befindet sich die Projektstruktur und das Feedback der Unit Tests. Das untere Fenster zeigt die Konsolenausgabe des ausgeführten Programms. [Gol12]

In seiner Arbeit evaluiert Goldman seine Entwicklungsumgebung Collabode anhand der genannten Modelle. Für das **Micro-Outsourcing** teilte er die Probanden in zwei Gruppen. Zunächst akquirierte er sieben Studenten und zwei professionelle Softwareentwickler, welche die Rolle des *Original Programmers* (Task-Distributor) einnahmen. Die Crowd rekrutierte er über Ausschreibungen auf der Freelancer Plattform *oDesk*[31]. Mithilfe eines weiteren Einstellungstest filterte er aus den 74 Bewerbern, 21 passende Teilnehmer heraus [Gol12].

[31] `https://en.wikipedia.org/wiki/Upwork` (30.4.2017)

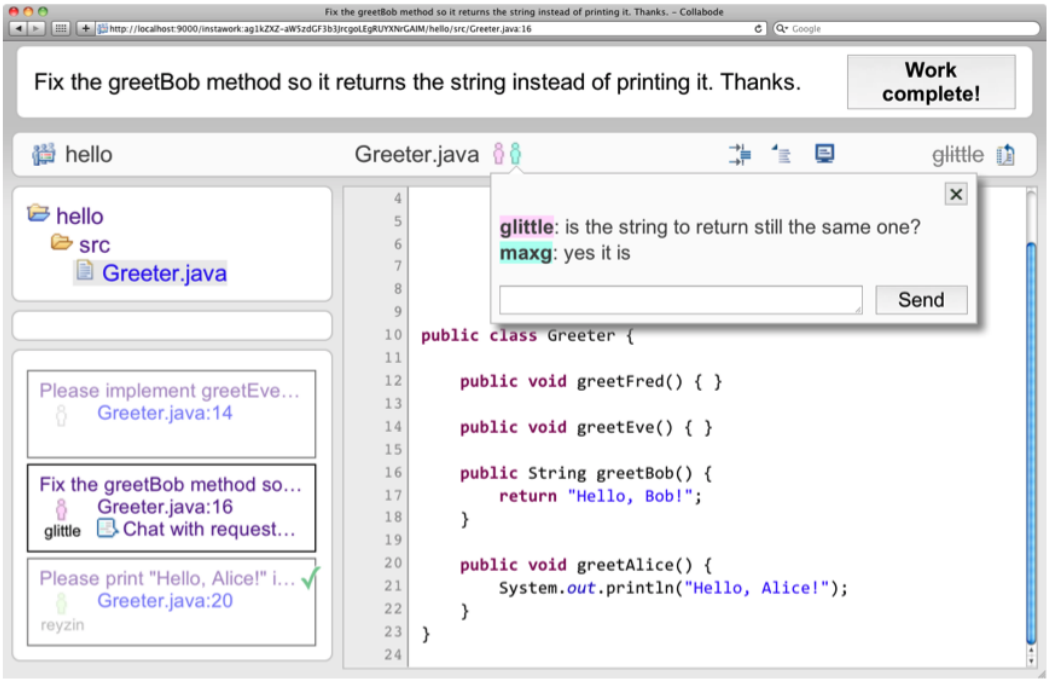


Abbildung 3.6.: Ansicht von Collabode, nachdem der Crowdworker die Aufgabe angenommen hat. [Gol12]

Für die Zuteilung der Mikroaufgaben entwickelte Goldman *Instawork*, eine in Python geschriebene Anwendung, welche Notifications über die Google Chat XMPP API an die Crowdworker senden kann [Gol12]. Durch die mitgesandte URL erhielt der Crowdworker Zugang zu Collabode, in welcher er die Aufgabe lösen sollte. Hierfür erweiterte Goldman die Anwendung um eine *Worker Ansicht* (Abbildung 3.6).
Zu Beginn der Evaluation wurde zunächst aus einem Pool von drei Programmierproblemen eines dem Task-Distributor zugeteilt [Gol12]. Im Anschluss sollte dieser die Aufgabe analysieren und eine Umsetzung planen. Hierfür legten die meisten zunächst ein Codegrundgerüst an, um anhand dessen die auszulagernden Aufgaben zu spezifizieren. Dabei entstanden verschiedenste Aufbereitungen für die Crowdworker. Einige Task-Distributoren arbeiteten den Gesamtkontext für die Aufgabenbeschreibung grafisch aus, andere fügten an ausgewählten Abschnitten des Quellcodes Kommentare hinzu. Um den Aufgabenbereich eines Crowdworkers einzuschränken, spezifizierten die Task-Distributoren meist wenige Funktionen, in denen die Lösungen durchgeführt werden sollten. Da Aufgabe und Funktionen im Code aber nicht verlinkt werden konnten, kam es zu Missverständnissen und Mehraufwand für die Crowdworker. Genauere Beispiele, wie detailliert die Aufgabenbeschreibungen formuliert wurden, beschreibt Goldman nicht. Der hohe angegebene Kommunikationsaufwand zwischen Task-Distributoren und Crowdworker lässt aber auf unzureichend aufbereitete Aufgaben schließen. Das erfasste Feedback der beiden Gruppen spiegelt jedoch ein eher zufriedenstellendes Ergebnis wider. Task-Distributoren merkten positiv an, dass sie sich eher auf die Planung des Softwareprojektes fokussieren konnten und so weniger Zeit mit *Low-Level-Implementationen* verbringen mussten. Weitere Aufgabensteller gaben positives Feedback, es fühle sich mehr nach Niederschreiben von Spezifikationen an und man könne sich so besser auf den *Flow* des Codes und die schwierigeren Abschnitte konzentrieren.

Als Implikationen für eine weitere Entwicklungsiteration der *Worker-Ansicht* von Collabode fasste Goldman folgende Punkte zusammen [Gol12]:

- detaillierter auf die zu bearbeitenden Stellen im Quellcode hinweisen
- Quellcode, Aufgabenstellung und Fragen stärker verbinden
- irrelevantes Feedback (zum Beispiel durch fehlerhafte Tests für andere Aufgabe) ausblenden
- die Implementierung einfacher überprüfbar machen, zum Beispiel mit ausführbaren Tests
- bessere Kommunikationsmöglichkeit zwischen Crowdworker und Task-Distributor anbieten

Die von Goldman gezogenen Schlüsse sollten auch in der Ausarbeitung des Konzepts dieser Arbeit beachtet werden. Zusammenfassend ist die Collabode Plattform ein interessanter Ansatz, um crowdbasierte Softwareentwicklung umzusetzen. Die ersten Evaluationen zeigen ein positives Interesse von Task-Distributoren und Crowdworkern. Bei der Aufbereitung und Spezifizierung der Microtasks sollten die Task-Distributoren aber deutlich mehr unterstützt werden. Hier fehlen dem Konzept klare Richtlinien und Toolunterstützungen.

3.3.2. CROWDCODE

CrowdCode ist eine Cloud-Entwicklungsumgebung für crowdbasierte Softwareentwicklung mit JavaScript [LBAV14]. Sie ist auf das parallele Bearbeiten von *Microtasks* zugeschnitten. Hierfür müssen Softwareprojekte zunächst in kleine atomare Aufgaben aufgespalten werden. Für die Aufteilung wird ein *Map-Reduce* Style Paradigma verwendet, welches im Rahmen dieser Arbeit nicht weiter beschrieben wird [KSK11].

Microtask type	Description	Possible subsequent microtasks
Write Function	Sketch or implement a function using code, pseudocode, and pseudocalls.	*Write Function, Reuse Search, Machine Unit Test*
Reuse Search	Given a pseudocall and the surrounding code, determine if an existing function provides the functionality or that no function does.	*Write Call, Write Function Description*
Write Function Description	Given a pseudocall and the surrounding code, write a description and signature for a new function for this behavior.	*Write Call*
Write Call	Replace the specified pseudocall with a call to the specified function or edit the function to implement the behavior in an alternative way.	*Write Function, Reuse Search, Machine Unit Test*
Write Test Cases	Given a description of a function, list test cases.	*Write Test*
Write Test	Given a test case and the description of a function, implement the test case or report an issue in the test case.	*Machine Unit Test, Write Test Cases*
Machine Unit Test	Executes all implemented tests, notifying functions if they fail a test	*Debug*
Debug	Edit code to fix bug, report an issue in a test, or create stubs describing issues in function call	*Machine Unit Test, Write Function, Reuse Search*

Abbildung 3.7.: Tabelle für die *Microtaskstypen* auf der CrowdCode Plattform [LBAV14].

CrowdCode spezifiziert Funktionen als zentrale Einheit der Granularität eines Softwareprojekts [LBAV14].
Zu diesen existieren verschiedene Aufgabentypen, welche in Abbildung 3.7 dargestellt werden. Um die Qualität der Crowdsourcing Arbeit zu sichern, müssen für alle Funktionen *Unit-Tests* geschrieben werden. Hierfür wird für jede *Write Function* Task eine *Write Test Cases* Aufgabe generiert.

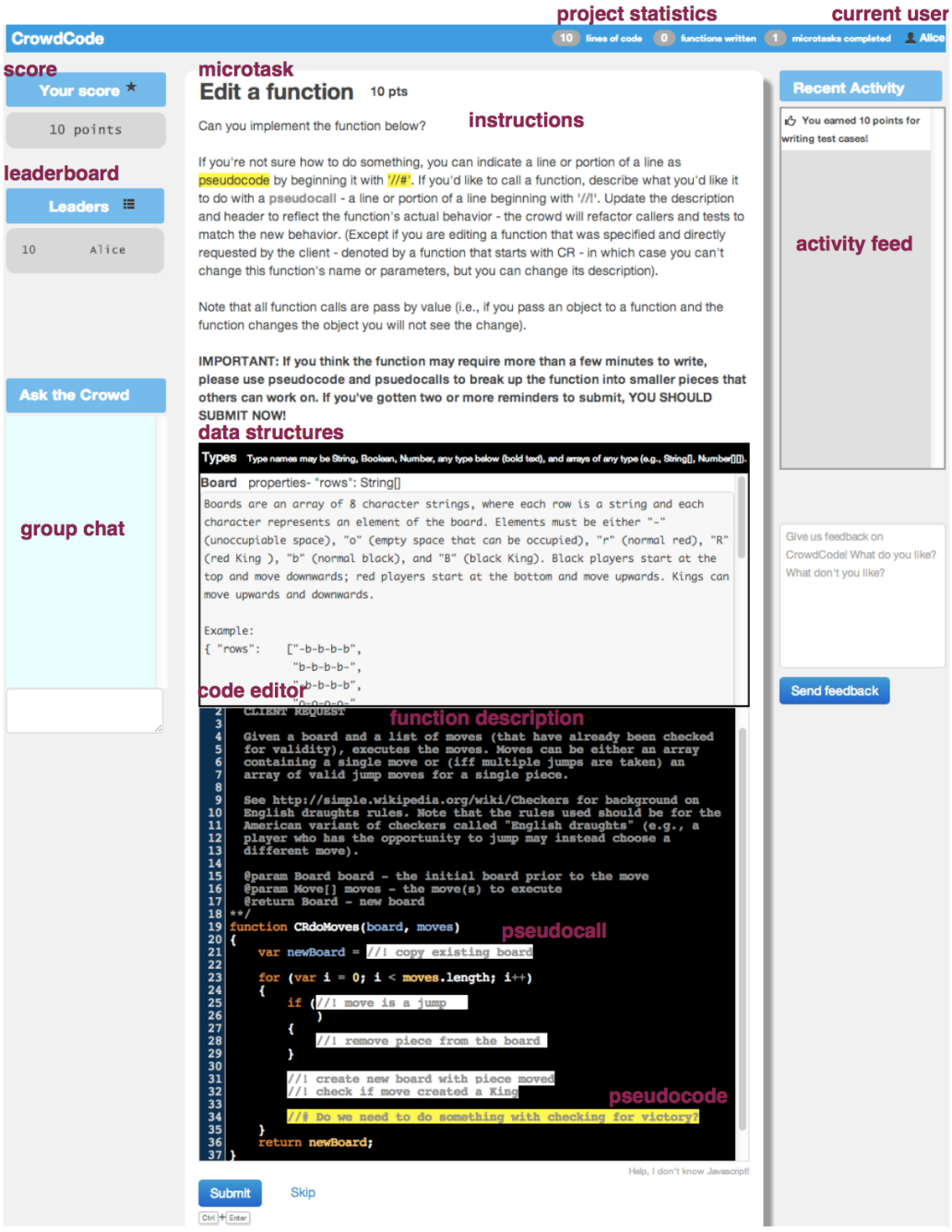


Abbildung 3.8.: Bearbeitungsansicht von CrowdCode nach Annahme einer Programmieraufgabe [LBAV14].

In jeder Microtask ist der Nutzer auf die Bearbeitung einer Funktion beschränkt. Abbildung 3.8 zeigt die Entwicklungsumgebung, in welcher die Lösung von *Write Function* Aufgaben statt findet. In dieser erhält der Nutzer zunächst Anweisungen, die er bei der Implementierung auf der Plattform berücksichtigen soll. So ist es dem Nutzer gestattet, neben Javascript Code, auch Pseudocode zu verwenden. Reicht ein Crowdworker Aufgaben mit Pseudocode ein, werden weitere Microtasks für die Implementierung generiert. Als Aufgabenbeschreibung dienen Kommentare oberhalb der Funktion. Zusätzlich müssen Task-Distributoren die Art der Übergabe- und Rückgabeparameter detaillierter erläutern (siehe Abbildung 3.8). Enthält die Implementierung einer Funktion keinen Pseudocode und alle zugehörigen Unit-Tests sind erfüllt, gilt die Microtask als abgeschlossen.
In einer ersten Nutzerstudie mit 12 Studenten der UCI (University of California, Irvine) konnte die CrowdCode Plattform erfolgreich evaluiert werden. Dabei erhielten die abgegrenzten Microtasks nicht durchgehend positives Feedback. Gerade deren Abgeschlossenheit und die daraus folgende Kontextunabhängigkeit wurden stark differenziert aufgenommen [LBAV14]. Einige Probanden merkten an, dass sie weitere Kontextinformationen zum Zustand des Systems erhalten wollen. Dieser Fakt ist verständlich, da es innerhalb einer Aufgabe möglich ist, andere Funktionen aufzurufen. Eine Person klagte über die ständige Änderung des mentalen Kontexts beim Wechsel zu neuen Microtasks.
Die Evaluation des Konzepts zeigt, dass selbst im Rahmen kleiner abgeschlossener Aufgaben, Crowd-Entwickler Informationen zum Kontext der Aufgaben und eine Übersicht über den Zustand des Gesamtsystems benötigen. Dies stimmt mit den Ergebnissen aus der Analyse des Programmverstehens (Kapitel 2.2) überein. Es fehlt eine Komponente, welche Crowdworker bei diesem Prozess unterstützt. Insgesamt ist die CrowdCode Plattform eine interessante Umsetzung von Crowd-basierter Softwareentwicklung. Speziell der testgetriebene Ablauf und die Möglichkeit, Pseudocode zu verwenden, sind gute agile Ansätze. Die Aufgabenbeschreibung in den Quelltext als ausführlichen Kommentar über die Funktion zu schreiben, führt zu Unleserlichkeit und erhöhten Wartungsaufwand des Quellcodes und sollte überdacht werden [Rob13].

3.4. FAZIT

Durch die State of the Art Analyse dieser Arbeit konnten die technischen Voraussetzungen für die Umsetzung eines eigenen Konzepts erarbeitet werden. Das *Language-Server-Protocol* bietet eine einfache Möglichkeit, unterstützende Editorfunktionalitäten sprachenunabhängig bereitzustellen.
Die *Eclipse Orion* Anwendung kann aufgrund der fehlenden Containerunterstützung nicht verwendet werden (Nicht alle Softwareprojekte wären ausführbar). Durch die entkoppelte Nutzbarkeit einzelner Bestandteile des Orion Projekts, kann aber der Editor für die Umsetzung verwendet werden. Da dieser in der *Eclipse Che* Anwendung erfolgreich mit dem Language-Server-Protokoll verwendet wird, ist eine Umsetzbarkeit gesichert. Für das Verwalten und Ausführen des Quellcodes bietet *Eclipse Che* alle benötigten Funktionalitäten an. Durch die ausführlich dokumentierten Schnittstellen ist eine Verwendung in der technischen Umsetzung des Konzepts ohne großen Aufwand möglich. Somit werden Teile der Anforderungen abgedeckt (Tabelle 3.1).
Die Analyse ähnlicher *Crowd-basierter Softwareentwicklungskonzepte* zeigt eine positive Grundstimmung gegenüber der geplanten Form der Entwicklung. Die genannten Probleme in den Evaluationen der Konzepte bestätigen die aufgestellten Anforderungen dieser Arbeit.

In *Collabode* führte die lose Kopplung zwischen Aufgabenstellung und Quellcode zu Missverständnissen und Mehraufwand. Weiterhin verursachte die freie Erarbeitung der Aufgabenbeschreibung (ohne Richtlinien) der Task-Distributor unklare Aufgabenstellungen. Dadurch kam es zu einem erhöhten Kommunikationsaufwand zwischen Task-Distributor und Crowdworker. Im eigenen Konzept muss der Task-Distributor mehr geführt und eingeschränkt werden, um eine gleichbleibende Qualität der Beschreibungen zu sichern. Die Evaluation von *CrowdCode* zeigte, dass auch bei Microtasks der Kontext einer Aufgabe nicht vernachlässigbar ist. Dem Crowdentwickler werden hierzu wenig Hilfen gegeben um sich in das bestehende Projekt einarbeiten zu können. Da der Nutzer aber Funktionen aus dem Programm wiederverwenden soll, müssen diese ihm zunächst vorgestellt werden. Die Ansätze von CrowdCode, dass nur eine Funktion vom Crowdentwickler bearbeitet wird, bedingte Microtasks automatisch erstellt werden, oder Funktionsaufrufe als Pseudocode geschrieben werden können, sind dahingegen positiv zu erwähnen.
Zusammenfassend ist anzumerken, dass durch die vorhergehende Analyse des Programmverstehens und des Kontextes (*testgetriebene Crowdsourcing-Plattform*) keine weiteren Anforderungen in der State of the Art Analyse hinzugekommen sind. Dafür konnten viele Anregungen für das eigene Konzept gesammelt werden.

Anforderung	Tool	Beschreibung
M20	Orion Code Edit Widget	Der extrahierte Editor von Eclipse Orion ermöglicht die Implementierung von Quellcode und Syntaxhervorhebungen. Weiterhin bietet er Schnittstellen für die Implementierung von Code Hilfen (z.B. Hover und Autocompletion).
M40	Eclipse Che	Der erarbeitete Quellcode wird in Workspaces von Eclipse Che gespeichert. Durch die Nutzung von Docker, können verschiedenste Anwendungen in isolierten Linux Umgebungen (Containern) ausgeführt werden. Mit Hilfe der Command Api von Eclipse Che, können Testausführungen angestoßen werden.
M50	Eclipse Che	Die Ausgabe des Quellcodes kann über eine Verbindung mit dem Workspace Agent von Eclipse Che ausgelesen werden.
S20, C20	Language-Server-Protokoll	Das Language-Server-Protokoll und Language-Server bieten die Grundlage für sprachenunabhängige Implementierungshilfen. Funktionalitäten wie Hover, Autocompletion und Go-To Aufrufe müssen nicht vom Editor Widget für jede Sprache einzeln implementiert werden.

Tabelle 3.1.: Abdeckung der technischen Anforderungen

4. KONZEPTION

Die Aufgabe dieser Arbeit besteht in der Entwicklung eines Programmiersystems, welches den Prozess Crowd-basierter Softwareentwicklung unterstützt. Hierfür soll die zu erarbeitende Lösung in einem Konzept einer testgetriebenen Crowdsourcing-Plattform für Softwareentwicklung einfließen und evaluiert werden können. Nach Vorstellung der Idee der Plattform in Abschnitt 4.1, wird das Konzept dieser Arbeit beschrieben. Dieses wird komponentenweise beschrieben.

4.1. POLYLITH - TESTGETRIEBENE CROWDSOURCING PLATTFORM

Polylith ist eine testgetriebene Crowdsourcing-Plattform für Softwareentwicklung und soll das Outsourcing von Softwareprojekten unterstützen. Abbildung 4.1 zeigt den Entwicklungszyklus mit der Plattform.
Zunächst erteilt ein Kunde, welcher eine Softwarelösung benötigt, dem Betreiber der Plattform einen Auftrag. Dieser wird durch Software-Architekten (in diesem Kontext polylith-Architekten benannt) analysiert und auf Umsetzbarkeit geprüft. Im Anschluss erarbeiten die polylith-Architekten Infrastruktur und Codebasis sowie *natürlichsprachliche Softwaretests* für die Abnahme der Aufgaben. Im nächsten Schritt werden die erarbeiteten Aufgaben den Crowdworkern über die Plattform zugänglich gemacht. Eine Aufgabe kann von mehreren Entwicklern gleichzeitig angenommen werden. Die Anzahl der Ausschreibungen wird von den polylith-Architekten bestimmt. Dadurch wird eine schnellere Bearbeitung und eine höhere Erfolgsquote gewährleistet.
Das Implementieren der Lösung findet direkt in der Plattform statt. Die dafür benötigte Entwicklungsumgebung soll im Rahmen dieser Arbeit entstehen. Dabei muss diese auf den Kontext testgetriebener Entwicklung und Lösung kleinerer Implementierungsaufgaben angepasst werden um den Crowd-Entwickler maximal zu unterstützen. Gleichzeitig bietet die Nutzung einer einheitlichen Entwicklungsumgebung die Möglichkeit, den Kommunikationsaufwand mit dem Task-Distributor zu verringern und das *Crowdworking* transparenter zu gestalten. Dabei wird der Entstehungsprozess des Quellcodes vom Editor aufgezeichnet und in einem gesonderten Datenformat gespeichert. Der Fortschritt der Lösung ist für den Crowdworker durch den testgetriebenen Entwicklungsprozess zu jeder Zeit ersichtlich. Sind alle Tests erfüllt, kann der Entwickler die Aufgabe abschließen und erhält die ausgeschriebene Vergütung. Anschließend wird die eingereichte Lösung nicht-funktional bewertet. Hierfür wird der Quellcode nach Lesbarkeit (redundanter Quellcode, Verschachtelungstiefe und Code Smells) bewertet.

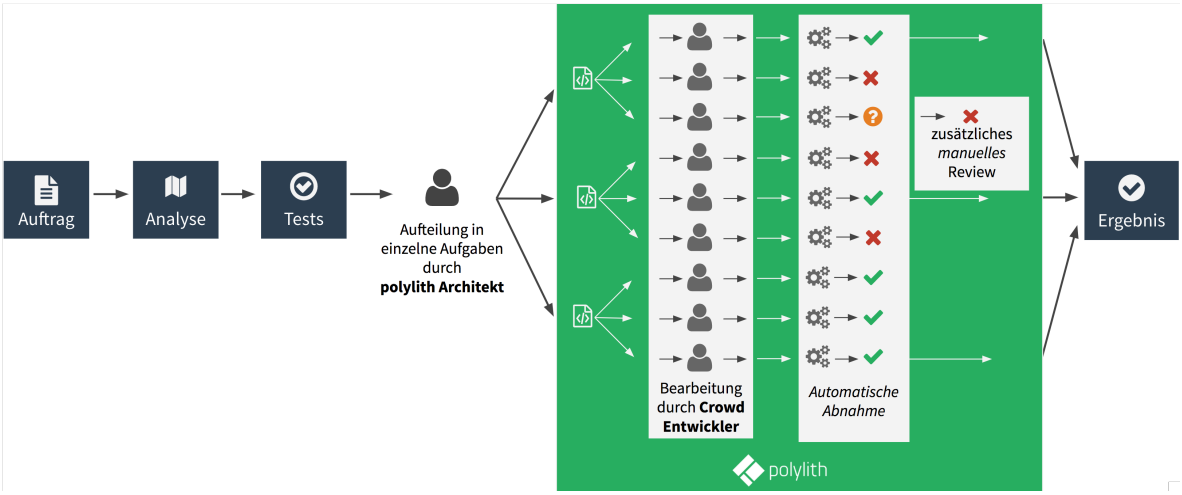


Abbildung 4.1.: Der polylith Entwicklungszyklus

Zusätzlich können durch die Aufzeichnung des Entstehungsprozesses der Lösungen weitere Metriken extrahiert werden [Blu17]. Hierbei wird dokumentiert, welche Tests zu welchen Zeitpunkt erfüllt waren. Wie in Abbildung 4.1 durch einen orangefarbenen Kreis mit Fragezeichen dargestellt, werden Auffälligkeiten im Quellcode vom System erkannt. Dabei kann es sich um Betrugsversuche oder eine Unterschreitung von Qualitätswerten handeln. Tritt dieser Fall auf, wird die abgegebene Lösung für ein manuelles Review vorgeschlagen. Hierfür bietet die Plattform eine speziell konzipierte Review-Ansicht, in der auffällige CodeAbschnitte hervorgehoben werden und der Entwicklungsprozess des abgegebenen Quellcodes nachvollziehbar dargestellt wird. Wurden mehrere Abgaben zu einer Aufgabe akzeptiert, wird die beste Lösung durch definierte Metriken ausgewählt. Weiterhin werden die extrahierten Metriken für ein Gamification-Konzept verwendet, um die Crowdworker stärker an die Plattform zu binden [Han17]. Zusätzlich ist geplant, dass die Plattform Bereiche aus den Abgaben zusammenführen kann, um die Qualität der Lösung weiter zu steigern. Im letzten Schritt findet die Aggregation der ausgewählten Lösungen statt. Zunächst wird dieser Schritt von den **polylith-Architekten** manuell durchgeführt. Ein teilautomatisierter Ablauf dieses Prozesses ist geplant.

4.1.1. EINBETTUNG IN DAS GESAMTSYSTEM

Abbildung 4.2 stellt die Einbindung dieser Arbeit in das Gesamtsystem der Plattform dar. Es setzt nach der Annahme einer Programmieraufgabe auf der Plattform an. Um diese zu bearbeiten, verpflichtet sich der Nutzer, diese vollständig in der ihm vorgegebenen Web-Entwicklungsumgebung zu lösen. Dies ist wichtig, um den Entstehungsprozess der Lösung aufzeichnen zu können und daraus wichtige Informationen über den Nutzer für die Plattform zu extrahieren. Sind alle Softwaretests bestanden und die funktionalen Anforderungen somit erfüllt, wird die Lösung durch die Qualitätskontrollkomponente der Plattform auf Lesbarkeit, Betrugsversuche und Performance geprüft [Blu17]. Wie in Abbildung 4.2 dargestellt, kann eine Aufgabe auf der Plattform von mehreren Personen angenommen werden. Zunächst ist nicht vorgesehen, dass diese kollaborativ an den Aufgaben arbeiten können, sondern jeder Nutzer eigenständig eine Lösung erarbeitet.

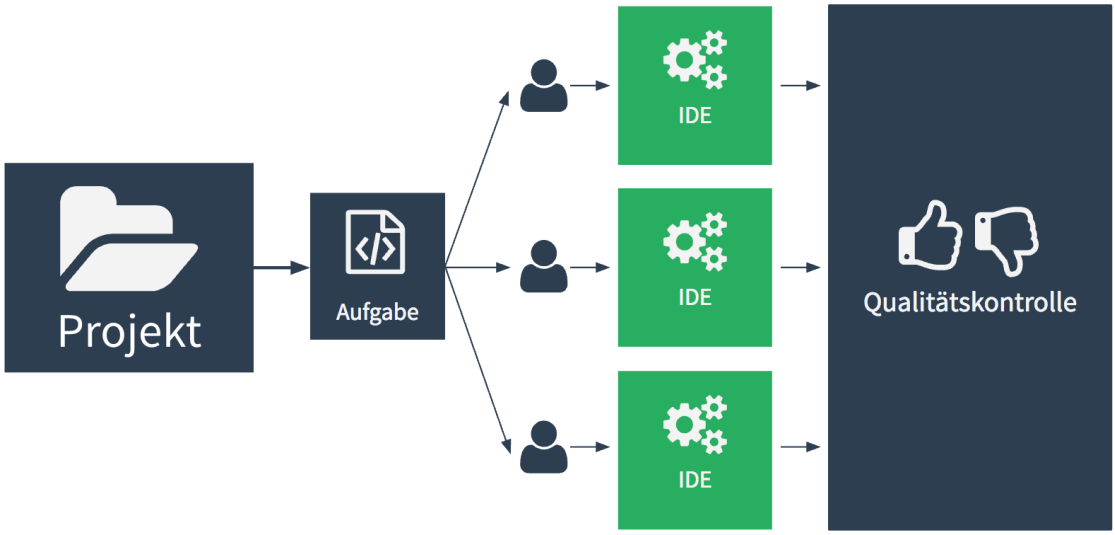


Abbildung 4.2.: Einbettung in das Gesamtsystem der Plattform (Konzept grün markiert). Die Entwickler arbeiten in getrennten Entwicklungsumgebungen.

Die Granularität der Aufgaben für die Crowdworker ist auf die Implementierung einer Funktion beschränkt. Es handelt sich um kleine funktionale Problemstellungen. User Interface Aufgaben sind zunächst nicht von der polylith-Plattform abgedeckt und werden im Konzept nicht betrachtet. Dies ähnelt dem Ansatz der *CrowdCode*-Plattform, welcher in Abschnitt 3.3.2 beschrieben wird. Folgende Informationen sind nach Annahme einer Aufgabe auf der Polylith-Plattform verfügbar:

Softwaretests: Natürlichsprachliche Sprachkonstrukte, die auf Quelltext abbilden. Diese werden vom Task-Distributor verfasst, um Aufgaben auf die Plattform zu stellen. Hierfür wird an einem Tool geforscht, welches ein sprachenunabhängiges Erstellen von natürlichsprachlichen Softwaretests ermöglichen soll. Die Grundlage für ein solches Werkzeug bildet die Arbeit von *Bauer*, in der Anforderungen für Programmiersprachen aufgestellt werden, um diese an natürlichsprachliche Softwarespezifikationen zu binden [Bau13].

Code-Grundgerüst des Projekts: Um die Ausführbarkeit der Softwaretests zu gewährleisten, muss ein ausführbares Code-Skelett des Softwareprojekts, in welchem die Aufgaben zu implementieren sind, vorhanden sein. Die Größe des Projekts ist dabei stark variabel.

Nutzerdaten: Informationen, die über Nutzer durch Bearbeitung anderer Aufgaben gesammelten werden konnten. Zum aktuellen Stand dieser Arbeit sind diese erfassten Metriken noch nicht festgelegt. Wie in der Anforderungsanalyse (Kapitel 2) beschrieben, ist der Prozess des Programmverstehen abhängig von der Programmiererfahrung des Bearbeiters. Mit den Entwicklermetriken wäre eine adaptive Anpassung der Hilfen und des Interfaces vorstellbar.

4.2. KONZEPT

Um einem Entwickler das Lösen einer Aufgabe in einer testgetriebenen Crowdsourcing Plattform für Softwareentwicklung zu ermöglichen, wird eine Ansicht bzw. Komponente benötigt, in der dem Nutzer das zugrundeliegende Softwaregerüst erläutert wird und gleichzeitig die Implementierung der Lösung stattfinden kann. Um die Anforderungen für den CrowdEntwickler dabei gering zu halten, erhält dieser stark abgegrenzte Aufgaben innerhalb eines Softwareprojekts. Als Aufgabenbeschreibung dienen natürlichsprachliche Tests. In diesen werden Sprachkonstrukte auf ausführbaren Quellcode abgebildet (Abbildung 4.3). Zunächst wird von Unit-

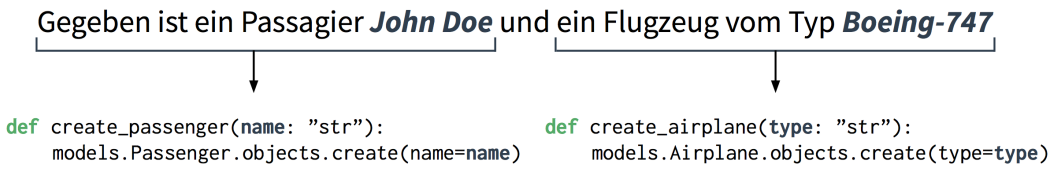


Abbildung 4.3.: Beispiel für natürlichsprachliche Tests in Python 3

Tests für abgegrenzte Funktionalitäten ausgegangen. Verwendung finden solche Spezifikationen im *Behavior Driven Development* (BDD), um eine bessere Kommunikation zwischen Domainexperte (Kunde) und Softwareentwickler zu ermöglichen [SWD12]. Diese Form der testgetriebenen Entwicklung soll auf der Crowdsourcing Plattform widergespiegelt werden, um von den Task-Distributor eine einheitliche Form der Aufgabenspezifikation zu fordern. Abbildung 4.4 zeigt den Behavior-Driven-Development Workflow. Die ersten drei Phasen übernimmt der Task-Distributor und stellt seine Lösung als Aufgabe auf die Crowdsourcing Plattform.

Abbildung 4.4.: Workflow des Behavior Driven Development. Grün hebt die Aufgabe des Crowd-Entwicklers hervor (abgeändert [SWD12]).

Die Grundidee des Konzepts dieser Arbeit ist es, die kognitive Belastung für den Crowdentwickler zu senken und so die Aufgabe leichter lösbar zu machen. Damit der Crowdworker die Implementierung erfolgreich durchführen kann, muss das Konzept die in Kapitel 2.3 aufgestellten Anforderungen erfüllen. Nach Annahme der Aufgabe befinden sich Crowdworker in einem unbekannten Softwareprojekt und Kontext. Wie die State of the Art Analyse von vergleichbaren Plattformen gezeigt hat, ist der Einstieg in die Aufgabe für den Nutzer eine schwere Hürde. Auch im Rahmen von kleinen funktionalen Aufgaben muss vorhandener Quellcode und der Kontext des zugehörigen Projekts dem Nutzer näher gebracht werden. Vor allem der häufige Wechsel zwischen unterschiedlichsten Domänen und Projekten ist für Nutzer eine starke kognitive Belastung (Kapitel 3.3.2). Hierfür muss die Anwendung den Entwickler nach Annahme der Aufgabe im Prozess des Programmverstehens unterstützen.

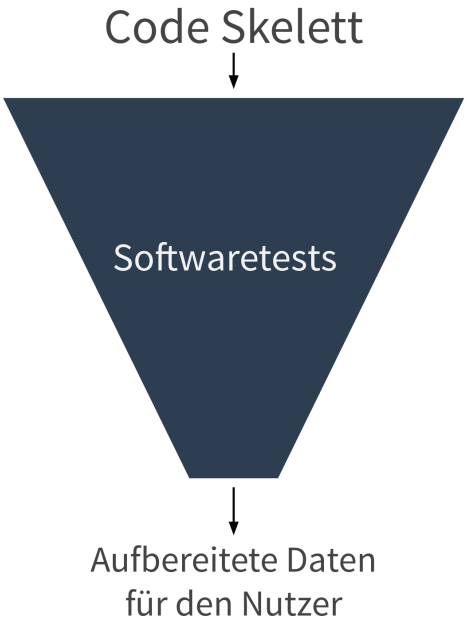


Abbildung 4.5.: Grundidee des Konzepts *Softwaretests* als Filter

Um das *Bottom-Up* Vorgehen zu unterstützen wird das Code-Grundgerüst mit Hilfe der vorliegenden Softwaretests des Task-Distributors untersucht und für den Crowdworker aufbereitet (siehe Abbildung 4.5). Durch statische Code-Analyse des Quellcodes der Tests können Code-Fragmente extrahiert werden, welche Abhängigkeit zu den Tests besitzen. Die Analyse könnte durch ein Tracking verbessert werden, welches den Task-Distributor bei der Implementierung der Tests analysiert. Dabei können Codeabschnitte und Dateien, die der Task-Distributor betrachtet oder verwendet, extrahiert werden. So kann der relevante Quellcode für den Entwickler eingeschränkt und die Informationsflut gesenkt werden. Weiterhin bietet es dem Nutzer einen leichteren Einstieg in das vorliegende Softwareprojekt. Reichen die angezeigten Codeabschnitte nicht aus, um die Lösung zu implementieren, muss die Anwendung eine Möglichkeit bieten, die Suche auszubreiten oder zusätzlichen Code anzufordern.
Für das *Top-Down* Vorgehen müssen die extrahierten Codeabschnitte in eine abstraktere Darstellung gebracht werden. Für eine erste Umsetzung ist ein vereinfachtes UML-Klassendiagramm geplant, welches für die Aufgabe relevante Modelle, Klassen oder Dateien darstellt. Da bei der Testanalyse Quellcode analysiert wird, ist dieser Standard zunächst am einfachsten zu generieren. Für weitere Iterationen sollten zusätzliche Abstraktionsmodelle untersucht werden. Der Entwickler muss dabei in der abstrakten Darstellung hinein- und herauszoomen können, um sich seine Position im Softwareprojekt deutlich zu machen.
Da die Umsetzung innerhalb eines Softwareprojektes stattfindet, der Crowdworker aber nur bestimmte Teile des Quellcodes einsehen kann, müssen seine Implementierung und die Softwaretests trotzdem ausführbar bleiben. Um dies zu ermöglichen muss eine klare Trennung zwischen Projektdaten und dem Editor des Crowd-Entwicklers vollzogen und eine Art Rechtesystem implementiert werden. Der Editor des Crowdworkers kann dabei als eine Art *Fenster* in das vorliegende Softwareprojekt verstanden werden. Durch die Remote-Verwaltung des Softwareprojektes hat der Nutzer keinen direkten Zugang zu Dateien und muss diese nicht auf seinem Rechner verwalten.

Dies erfüllt ebenfalls Forderungen auslagernder Unternehmen, welche nicht ihre komplette Entwicklung fremden Personen zugänglich machen wollen[1]. Um eine abgegebene Lösung möglichst nah an der Produktivumgebung des Task-Distributor zu testen, erhält dieser die Möglichkeit, durch Virtualisierung eine realistische Laufzeitumgebung für die Ausführung der Softwaretests zu schaffen. Die Nutzung einer solchen Technologie ist essentiell, um verschiedenste Softwareprojekte und Programmiersprachen abzudecken. Abbildung 4.6 fasst die geplanten Module, welche für die Arbeitsumgebung (*Workspace*) eines Crowd-Entwickler bereitgestellt werden, zusammen.

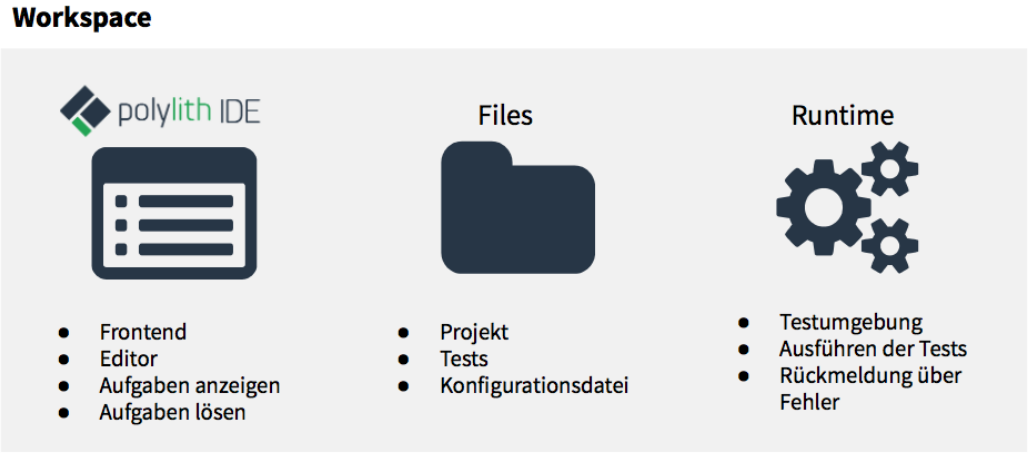


Abbildung 4.6.: Komponenten eines Workspaces, beinhaltet die nötigen Ressourcen und Funktionalitäten, um die Mitarbeit des Crowdworkers zu ermöglichen

4.2.1. KOMPONENTEN

Abbildung 4.7 zeigt die benötigten Komponenten, um die in Kapitel 2.3 aufgestellten Anforderungen zu erfüllen. Die Komponenten wurden dabei in die drei erarbeiteten Entwicklungsphasen eingeteilt, um eine möglichst vollständige Abdeckung der Anforderungen zu sichern.

Abbildung 4.7.: Übersicht über die benötigten Komponenten der Entwicklungsumgebung.

[1] https://de.statista.com/statistik/daten/studie/182667/umfrage/ (13.8.2017)

TITEL UND BESCHREIBUNG

Wie bei den in Kapitel 3 vorgestellten Plattformen muss eine Aufgabe einen Titel und eine Beschreibung besitzen. Diese sollten vom Task-Distributor prägnant gewählt sein. Zunächst ist geplant, dass die Beschreibung den natürlichsprachlichen Tests entspricht. Ob zusätzliche Beschreibungen benötigt werden, zeigt sich erst nach einer ersten Evaluation. Theoretisch ist es ebenfalls möglich, nicht jeden Satz mit einem Softwaretest zu hinterlegen, um so ausführliche Beschreibungen der Aufgabe zu erstellen.

TESTS

Bevor Aufgaben auf die Plattform gestellt werden können, müssen zur Sicherung der fachlichen Qualität Softwaretests erstellt werden. Gleichzeitig benötigt der Crowdworker detailierte und klar spezifizierte Aufgabenstellungen.
Die Beschreibung von Softwareartefakten mithilfe domänenspezifischer Sprachen (DSL) ist seit langer Zeit Gegenstand der Forschung und hat eine lange Tradition am Softwaretechnologie-Lehrstuhl der TU Dresden[2]. Insbesondere das Verfassen von Sofwaretests mithilfe natürlichsprachlicher Sätze ist bereits durch verschiedene Softwareprodukte ohne großen Aufwand möglich und hat sich in der Praxis weit verbreitet. In verwandten Crowdsourcing Projekten, wurden solche Spezifikationen noch nicht verwendet. Da die natürlichsprachlichen Sätze mit ausführbaren Softwaretests hinterlegt sind, entspricht Quellcode, der diese Tests erfüllt, stets der Aufgabenspezifikation. Dies bringt beim Outsourcing den Vorteil, dass der Task-Distributor die Beschreibung gleichzeitig als Anforderungen und Abnahmekriterien nutzen kann. Um die Informationsflut für den Nutzer gering zu halten, ist der Test-Quellcode für den Crowdworker nicht zugänglich.

NAVIGATIONSKOMPONENTE

Die Navigationskomponente muss den Nutzer beim Programmverstehen unterstützen und ihm gleichzeitig deutlich machen, in welchem Bereich des Projekts er sich befindet. Durch Analyse verwandter Arbeiten stellte sich heraus, dass eine solche Komponente den Crowdworkern fehlt. Aus diesem Grund werden extrahierte Klassen und Modelle aus den Tests in einem vereinfachten UML-Diagramm dargestellt. Diese Elemente werden mit der Editor Komponente verbunden. Über eine Tastenkombination muss der Entwickler aus dem Editor zusätzliche Informationen in der Navigationskomponente erhalten. Dafür werden die angewählten Modelle farblich hervorgehoben. Weiterhin muss der Nutzer eine Auflistung aller, für die Aufgabe verfügbaren, Dateien erhalten. Diese werden über eine statische Analyse der Tests extrahiert und einer Liste hinzugefügt. Da der Fokus dieser Arbeit zunächst auf der ersten Umsetzung des Gesamtkonzeptes liegt, sollte für die genaue Konzipierung der Navigationskomponente eine weitere State of the Art Analyse für *Reverse Engineering* durchgeführt werden. Einen Einstieg in dieses Themengebiet bieten die in Kapitel 2.2 genannten Arbeiten.

[2] `https://tu-dresden.de/ing/informatik/smt/st` (5.6.2017)

EDITOR

Der Code-Editor ist neben der Navigation die wichtigste Komponente des Konzepts. Dieser dient zur Umsetzung bzw. Implementierung der vorliegenden Aufgabe. Es existieren unzählige verschiedene Editoren und jeder Softwareentwickler hat seine eigenen Vorlieben. Es ist nicht möglich, eine perfekte Lösung für alle Anwendungsfälle und Präferenzen der Nutzer umzusetzen. Aus diesem Grund fokussiert sich der Editor dieser Arbeit zunächst auf den Kontext *testgetriebener* Entwicklung und *Bearbeitung kleiner Aufgaben auf Funktionsebene*. Zusätzlich sollten die Interaktionsmöglichkeiten auf das Nötigste beschränkt werden, um die Funktionsüberflut einzuschränken und Komponente iterativ nach den Anforderungen der Entwickler zu erweitern. Weiterhin soll der Editor programmiersprachenunabhängig funktionieren, um die Aufgabenvielfalt auf der Plattform nicht einschränken zu müssen.
In Collabode oder Crowdcode werden zusätzliche Informationen als Kommentare in den Quellcode geschrieben. Wird dieser an Unbekannte weitergereicht, neigen Entwickler zu sehr ausführlichen Kommentaren, um möglichst detailliert die Aufgabe zu beschreiben. In CrowdCode wird dabei sogar die Aufgabenstellung als Kommentar über die Funktion geschrieben. Dies führt zu unleserlichem und überfülltem Quellcode[Rob13].
Um Crowdworkern zusätzliche Informationen liefern zu können, muss dem Task-Distributor eine alternative Lösung angeboten werden. Über eine *Konfigurationsdatei* kann er deshalb zusätzliche Informationen zur Aufgabe hinzufügen. Dabei kann er durch Angabe des Namens eines Modells oder einer Funktion, das Element mit aufgabenspezifischem Wissen anreichern. Diese zusätzlichen Informationen sind anschließend in der *Hover*- und *Autocompletion*-Funktionalität eingebettet und erweitern die Informationen, welche diese Funktionalitäten standardmäßig liefern. Die Konfigurationsdatei ermöglicht somit das Hinzufügen von aufgabenspezifischen und kontextabhängigen Inhalten, ohne den Quelltext zu beeinflussen. Die angereicherten Daten werden darüber hinaus in die abstrakte Darstellung der Navigationskomponente hinzugefügt. Über die Konfigurationsdatei kann der Task-Distributor weiterhin Funktionen und Modelle angeben, die dem Nutzer bei der Bearbeitung der Aufgabe nicht angezeigt werden sollen. Er kann hierbei zwischen komplettem Ausblenden oder Darstellung des Methodenrumpfes wählen.

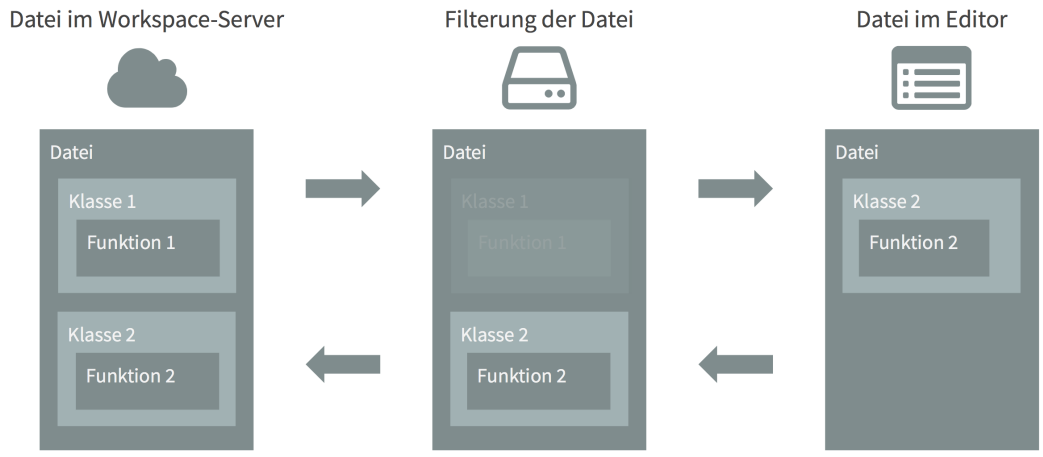


Abbildung 4.8.: Filterung der zu bearbeitenden Dateien durch eine Proxy-Komponente

Dies ermöglicht es dem auslagernden Unternehmen, wichtige Abschnitte ihres Programms intern zu behalten. Durch die Einschränkungen im Quellcode kann der Nutzer damit noch stärker gelenkt werden. Da beim Crowd- bzw. Outsourcing jeglicher zusätzlicher initialer Aufwand für die Aufgabenauslagerung abschreckend wirkt, ist das Anlegen einer *Konfigurationsdatei* keine Pflicht [EPS14].
Um den Nutzer weiterhin bei der Implementierung zu unterstützen, wird die Funktion der Codevorschläge auf das testgetriebene Setting angepasst. Hierfür erhält der Nutzer für die Lösung relevante Vorschläge von Modellen und Funktionen. Da er nur einen kleinen Teil des Softwareprojektes bearbeitet, werden ausgeblendete Abschnitte nicht in die Vorschlagsliste aufgenommen. Ermöglicht wird diese Funktionalität durch die statische Analyse der vorliegenden Softwaretests. Da der Editor nicht mit Inhalten überladen werden soll, können zusätzliche Anmerkungen zu Funktionen über das Hovern der gewünschten Objekte angezeigt werden. Weitere Daten über Abhängigkeiten und Positionierung im Projekt können durch eine einfache Tastenkombination innerhalb der Navigationskomponente aufgerufen werden. Dies ist wichtig, um die Verknüpfung zwischen den Phasen Implementierung und Analyse zu schaffen und somit den erarbeiteten Anforderungen gerecht zu werden (Anforderung C30).
Der Editor unterscheidet zwischen lesbaren und zu bearbeitenden Dateien. Funktionen und Klassen, die der Nutzer in den ausgewählten Files nicht sehen oder bearbeiten darf, werden nicht dargestellt (siehe Abbildung 4.8). Diese Funktionalität soll den Navigationsaufwand und damit gleichzeitig die kognitive Belastung des Nutzers senken.
Abbildung 4.9 visualisiert ein Beispiel der geplanten Editor-Komponente. Dabei wird die Methode *create_passenger* für den Crowd-Entwickler ausgeblendet. Um dem Betrachter die Bedeutung der *create_ticket* im Kontext zu erläutern wird diese mit Kontextinformationen angereichert.

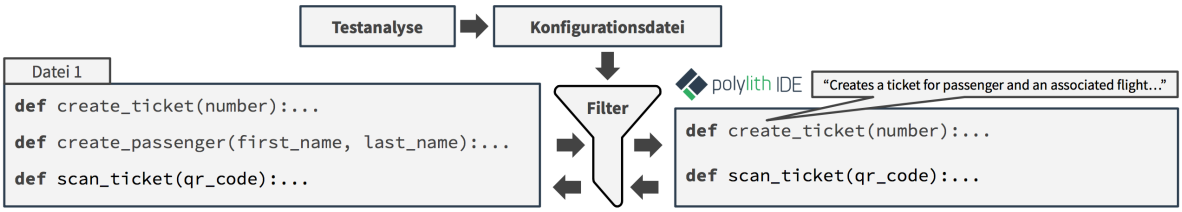


Abbildung 4.9.: Beispiel der geplanten Filterung und Anreicherung des Quelltextes. Die zusätzlich dargestellten Kontextinformationen kann der Nutzer über die Auto-Completion oder Hover-Tooltips erhalten.

FEHLERAUSGABE

Damit der Nutzer stetig Feedback über seinen Lösungsfortschritt erhält, muss sein implementierter Quellcode ausführbar sein und durch Softwaretests validiert werden. Zunächst ist eine automatische Ausführung der Tests im Hintergrund vorgesehen. Da diese serverseitig ausgeführt werden, wird die Performance des Editors auf Clientseite nicht beeinträchtigt. Entscheidend ist, dass die sich ständig ändernde Fehlerausgabe möglichst unauffällig im Userinterface visualisiert wird. Sollte sich diese Methode als störend für den Crowdworker herausstellen, muss eine manuelle Ausführung der Tests ermöglicht werden. Zunächst war eine Fehlerkonsole für die Visualisierung der fehlschlagenden Tests vorgesehen. Da die natürlichsprachlichen Softwaretests aber auf der Plattform erscheinen und der Lösungsprozess (siehe Abbildung 4.10) zeigt, dass Feedback und Analyse der Spezifikationen zusammenhängen, sollten Fehlerausgabe und Tests visuell verbunden werden.

KOMMUNIKATIONSTOOL

Die Hilfe-Komponente ist essentiell für die Plattform und gleichzeitig ein Problem für den TaskDistributor. In Softwaretests können wie bei anderen Implementierungen Fehler auftreten. Gleichzeitig kann es passieren, dass der Crowdworker Zugang zu weiteren Bereichen des Softwareprojektes erhalten muss. In beiden Fällen kann der Crowdworker seine Lösung nicht korrekt umsetzen oder das Ergebnis wird verfälscht. Tritt dieser Fall häufiger auf, sinkt die Schwelle des Nutzer, Fehler melden zu wollen. Trotzdem ist es wichtig, dass der Crowdworker diese Fehler möglichst einfach melden kann, um eine weitere Bearbeitung der Aufgabe zu ermöglichen. Damit der Task-Distributor die gemeldeten Probleme schnell lösen kann, erhält er Zugang zu der aktuellen Implementierung des Nutzers in einer eigenen Entwicklungsumgebung. Um keinen inkonsistenten Quellcodestand zu erhalten, muss der Task-Distributor vor der Bearbeitung die Aufgabe für den Nutzer temporär sperren.

4.2.2. INTERFACE GESTALTUNG

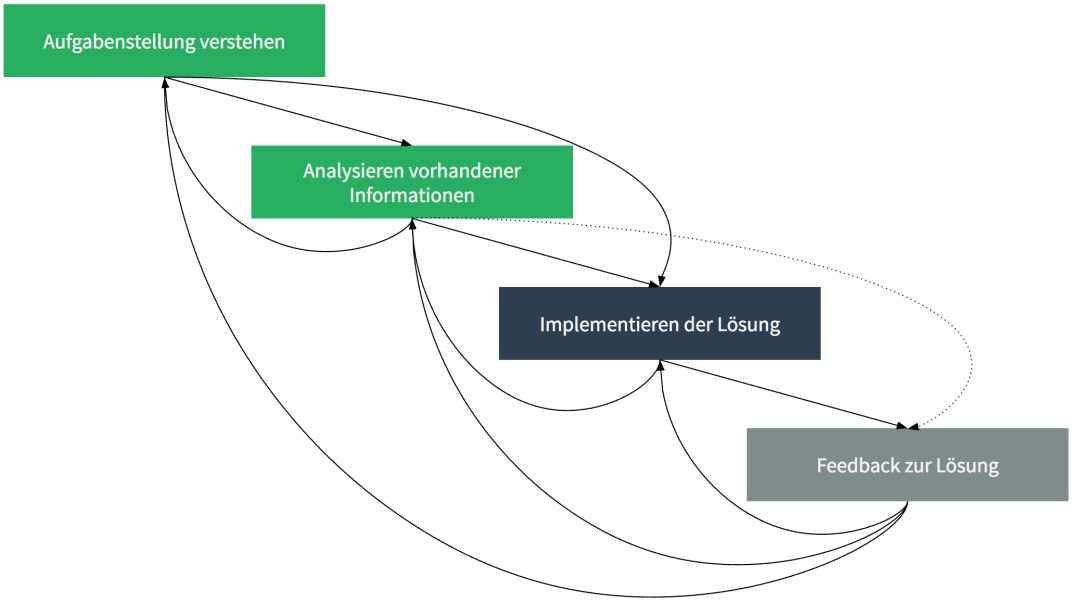


Abbildung 4.10.: Entwicklungsprozess auf der Plattform

Im folgenden Abschnitt werden erste Entwürfe für die Anordnung und Gestaltung des User-Interfaces vorgestellt. Dafür sollten frühzeitig Kenntnisse über potentielle Benutzer und ihre Aufgaben geschaffen werden [Pre10]. Um die Anordnung und Interaktion der Komponenten auszuarbeiten, wird im Folgenden der Ablauf bei der Bearbeitung einer Aufgabe auf der Plattform betrachtet. Durch die HTA-Analyse in Kapitel 2.3 wurde festgestellt, dass der Lösungsprozess iterativ ist. Wie in der testgetriebenen Entwicklung üblich, sollte der Nutzer versuchen, die vorgegebenen Tests nach und nach zu erfüllen [Bec03]. Kann er den Softwaretest nicht auf Anhieb lösen, analysiert er ein weiteres Mal das vorliegende Projekt und die Aufgabenstellung. Abbildung 4.10 zeigt diesen Ablauf grafisch. Da die einzelnen Phasen eng miteinander verknüpft sind und ein Wechsel einfach möglich sein sollte, müssen die unterstützenden Komponenten kompakt und auf einer Seite dargestellt werden. Gleichzeitig kann so der Navigationsaufwand innerhalb der Anwendung gering gehalten werden.

Die Implementierungsphase ist das Bindeglied zwischen Analyse und Feedback und für den Nutzer unumgänglich. Eine Ausnahme repräsentiert die gestrichelte Linie in Abbildung 4.10. Diese Verbindung müsste bei Crowdsourcing-Aufgaben betrachtet werden, welche keine Implementierungen fordern. Da solche Aufgabentypen zunächst nicht von der Plattform abgedeckt werden, wird diese Verbindung für die Konzeption der Benutzerschnittstelle nicht betrachtet. Ein solcher Aufgabentyp benötigt weiterhin zusätzliche Komponenten.

ENTWURF 1

Im ersten Entwurf (Abbildung 4.11) werden die Komponenten in die Crowdsourcing Plattform eingebettet. Die Anordnung der Bausteine orientiert sich dabei stark an der Reihenfolge, des in Abbildung 4.10 dargestellten Verlaufs.
Wie auch in verwandten Applikationen werden zunächst Titel und Beschreibung der Aufgabe angezeigt. Im Anschluss folgt die Darstellung der Tests und Navigation. Diese drei Elemente decken die Analysephase ab. Da nicht angenommen werden kann, dass keine zusätzlichen Beschreibungen zu den natürlichsprachlichen Softwaretests benötigt werden, existiert zwischen beiden Komponenten eine klare Trennung. Der Editor nimmt als zentrale Komponente die gesamte Breite der Webapplikation ein. Insgesamt ist der Aufbau sehr ähnlich zu den untersuchten Online Editoren.
In einer ersten schnellen prototypischen Umsetzung wurden einige Usability-Probleme dieser Anordnung durch Diskussion mit drei Probanden aufgedeckt. Zunächst müsste die Test Komponente bei einer großen Anzahl an Tests scrollbar sein und nicht alle Tests wären auf einen Blick sichtbar.

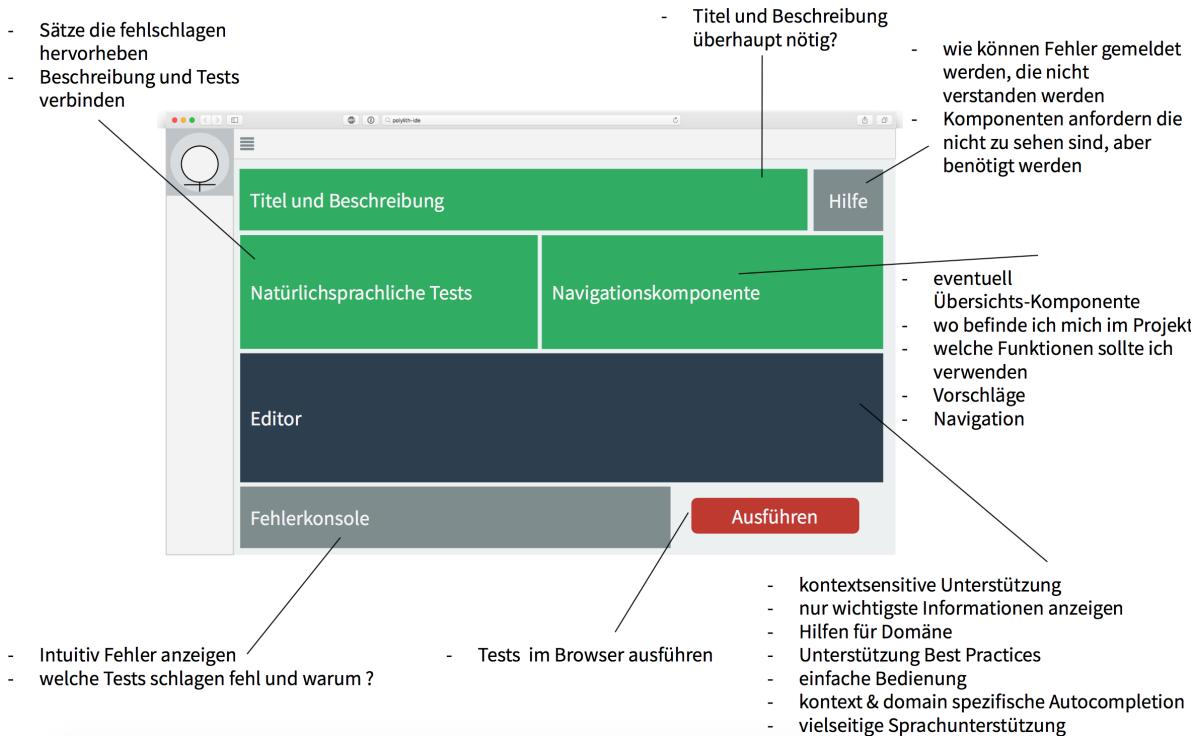


Abbildung 4.11.: Erster Sketch des User-Interfaces mit Anmerkungen und Überlegungen zu den einzelnen Komponenten

Gleiches gilt für die Navigationskomponente, die bei einer gewissen Anzahl von freigegebenen Dateien skalierbar sein muss. Da die Softwaretests in natürlicher Sprache auf die Plattform gestellt werden, muss die Fehlerkonsole auf Sätze verweisen. Die Zuordnung ist somit abhängig von der Ausgabe der Fehlerkonsole und Nutzer müssten zwischen den Komponenten hin und her springen. Aus diesem Grund entstand die Idee, Fehlerkonsole und natürlichsprachliche Tests zu kombinieren, wie es in der Auflistung der Komponenten beschrieben wird (Kapitel 4.2.1). Aufgrund der genannten Probleme wurden weitere Sketches erarbeitet.

ENTWURF 2

Im zweiten Sketch (Abbildung 4.12) wurden zunächst die Komponenten Titel und Beschreibung verkleinert, da angenommen wird, dass die natürlichsprachlichen Tests möglichst ausführlich und mit vielen Kontextinformationen ausformuliert sind. Um den Aufwand durch Scrollen innerhalb der Komponenten (Natürlichsprachliche Tests und Navigationskomponente) zu minimieren, nehmen diese nun alle die gesamte Breite der Anwendung ein. Da heutzutage Breitbild-Monitore den Großteil der verwendetet Displays ausmachen, sollen die Inhalte möglichst horizontal verteilt werden. Das Feedback nach Ausführung der Tests soll direkt in den natürlichsprachlichen Tests dargestellt werden. Aus diesem Grund wurde die Fehlerkonsole entfernt. Abbildung 4.13 zeigt verschiedenste Sketches für das Fehlerfeedback. Die finale Version soll in Zusammenarbeit mit den Probanden zu einem späteren Zeitpunkt erarbeitet werden. Ein weiterer Unterschied zum ersten Entwurf ist der Verzicht auf die Ausführungskomponente. Diese Entscheidung wurde unter der Annahme getroffen, dass die Tests im Hintergrund ausgeführt werden, sobald der implementierte Quellcode eine valide Syntax besitzt. Dabei muss die Benutzeroberfläche dem Nutzer verdeutlichen, wann die Tests ausgeführt werden und wann der Testdurchlauf fertig ist. Vorstellbar hierfür wären ein Ladebalken oder ein rotierender Kreis. Ob der Nutzer berechtigt werden sollte, zwischen automatischer und manueller Ausführung umschalten zu können, muss in einer ersten Evaluation analysiert werden.

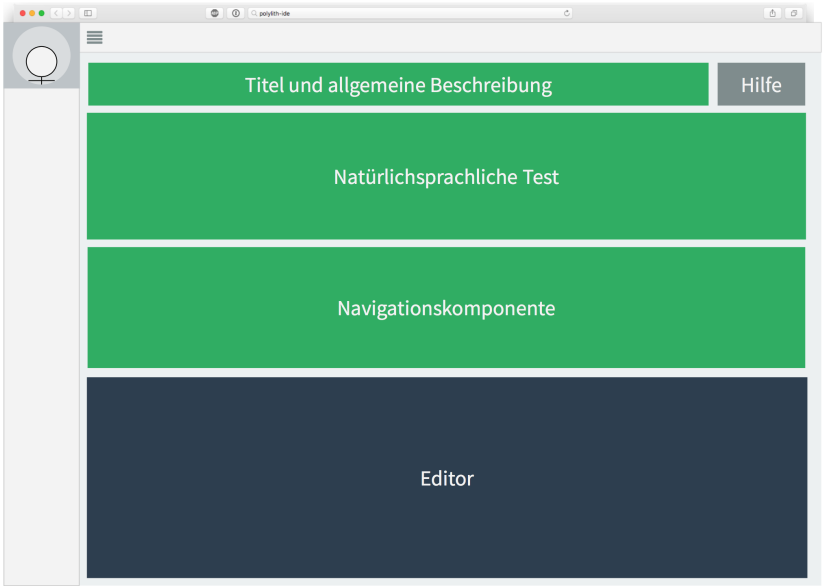


Abbildung 4.12.: Zweiter Sketch des User-Interfaces

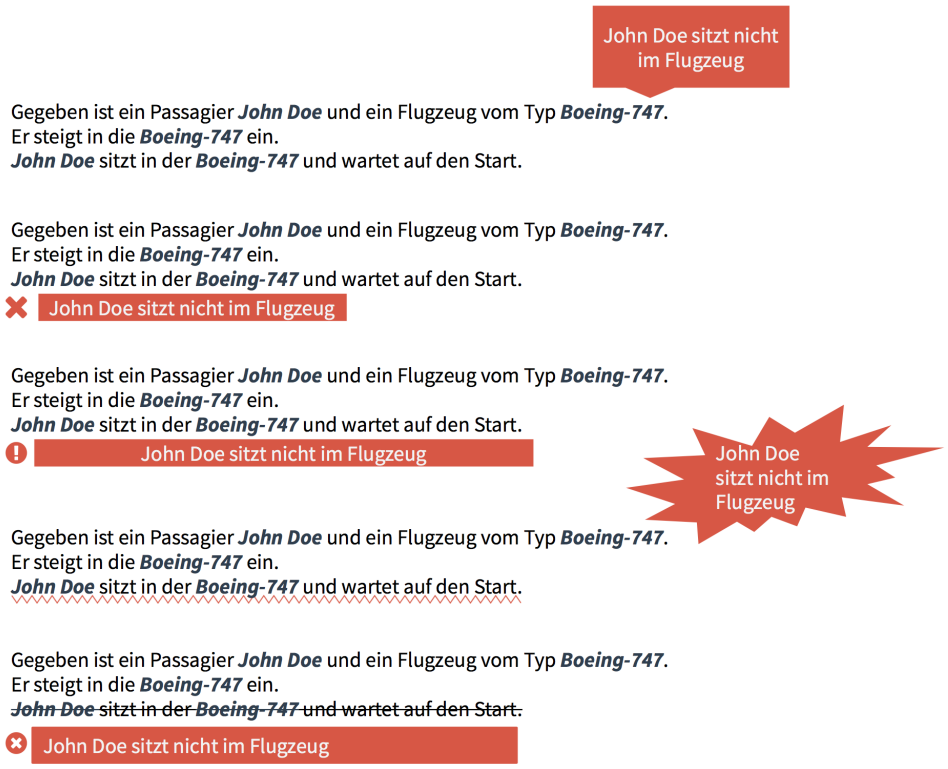


Abbildung 4.13.: Sketching der Visualisierung von Fehlern innerhalb der natürlichsprachlichen Tests

ENTWURF 3

Der dritte Sketch (Abbildung 4.15) bildet die Grundlage für die Umsetzung des ersten Prototypen. Die Editor Komponente und die natürlichsprachlichen Tests werden in dieser Lösung nebeneinander platziert. Diese Entscheidung wurde aufgrund der HTA-Analyse (Kapitel 2.3) getroffen. Die Positionierung wie im zweiten Entwurf ermöglicht keinen direkten Sprung von Implementierung zurück zum Aufgaben (Test) lesen. Gerade bei langen Aufgabenbeschreibung und vielen Inhalten in der Navigationskomponente führt dies zu einem hohen Navigationsaufwand für den Benutzer. Weiterhin sollten der Editor und die natürlichsprachlichen Tests skalierbar sein. Der Nutzer kann so den Fokus zwischen Bearbeitung und Lesen der Aufgabe einstellen.

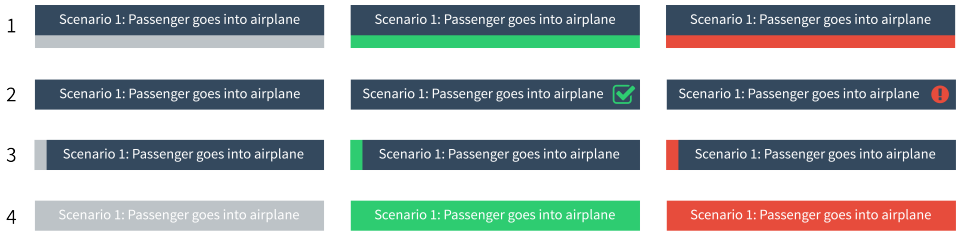


Abbildung 4.14.: Darstellungen für nicht ausgeführte, bestandene und fehlgeschlagene Test-Szenarios.

Abbildung 4.14 zeigt weiterhin verschiedene Visualisierungsmöglichkeiten für den Status der Test Szenarios. Für die Umsetzung sollte Sketch **2** oder eine Kombination mit einem weiteren Mock-Up verwendet werden, da ansonsten die Barrierefreiheit eingeschränkt wird (Rot-Grün Schwäche).

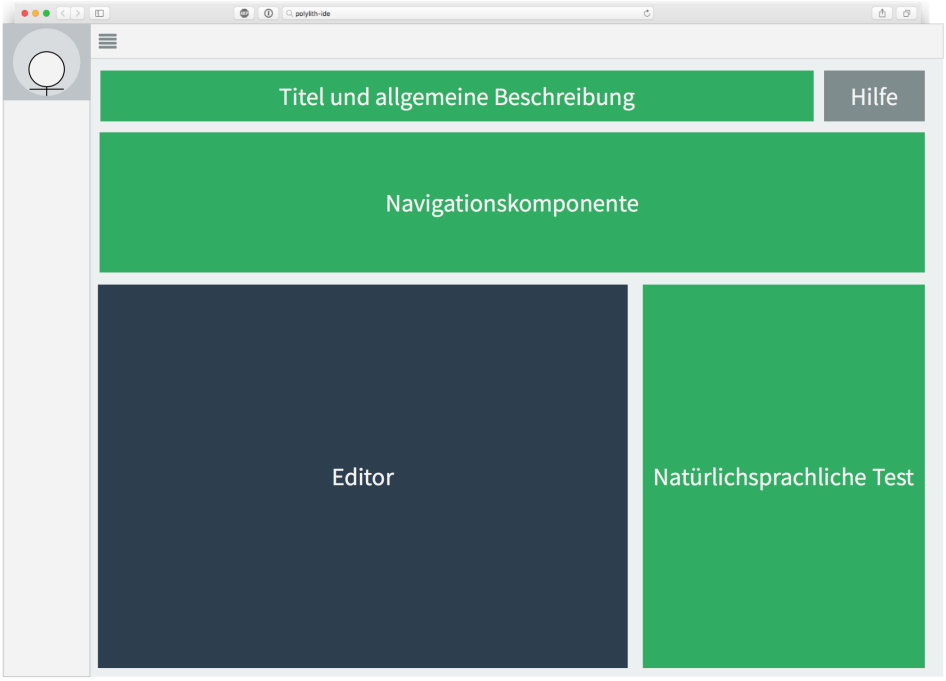


Abbildung 4.15.: Dritter Sketch für die Benutzerschnittstelle

5. UMSETZUNG

Das folgende Kapitel beschreibt die geplante technische Umsetzung und die prototypische Implementierung des in Abschnitt 4.2 beschriebenen Konzepts. Zunächst werden die benötigten Komponenten genannt und die grobe Umsetzungsidee beschrieben. Anschließend wird die geplante Infrastruktur vorgestellt, um die Kommunikation zwischen den einzelnen Modulen näher zu beschreiben. Im Anschluss wird detaillierter auf die Implementierung der einzelnen Komponenten eingegangen.

5.1. BENÖTIGTE KOMPONENTEN

Um das in Kapitel 4.2 vorgestellte Konzept umzusetzen, werden neben den einzelnen Modulen der Entwicklungsumgebung (**polylith-IDE**) weitere Komponenten benötigt. Abbildung 5.1 zeigt die Unterteilung in drei große Bausteine.

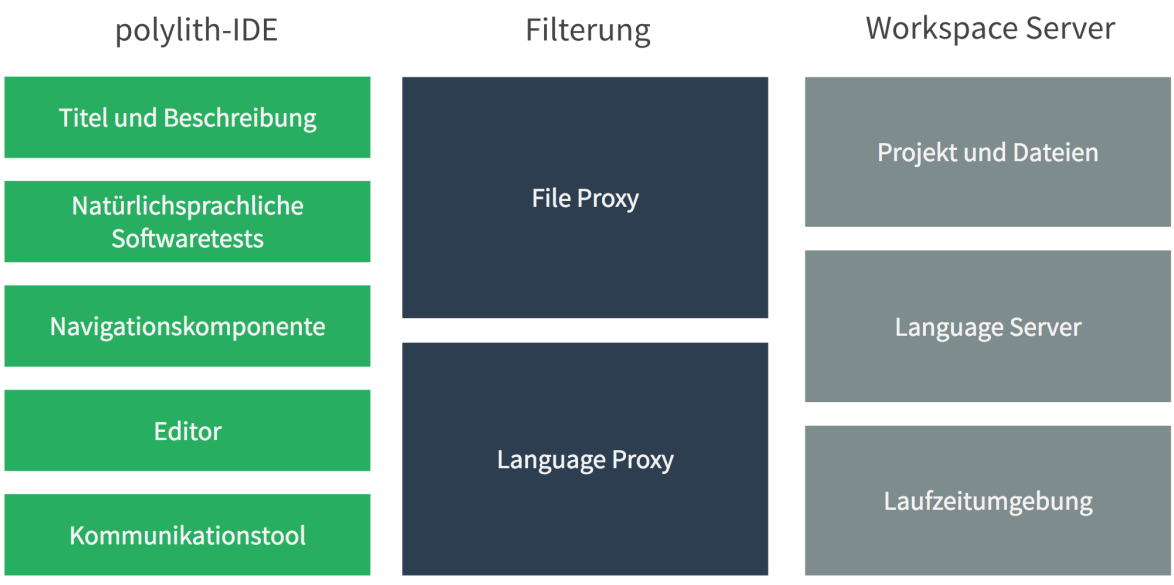


Abbildung 5.1.: Benötigte Komponenten für die Umsetzung des Konzepts

Die **polylith-IDE** beinhaltet die Entwicklungsumgebung, in der Crowdworker ihre Lösungen implementieren und ist die Frontend-Komponente der Client-Server Anwendung. Die zugehörigen Bausteine müssen in eine Webplattform integrierbar sein und möglichst unabhängig voneinander arbeiten. Die Umsetzung von Titel und Beschreibung sind trivial. Natürlichsprachliche Softwaretests erfordern eine Renderingkomponente, welche die Spezifikationen in ein HTML-Format transformiert und gleichzeitig ein Mapping mit der Fehlerausgabe ermöglicht. Die Navigationskomponente sollte ein Datenformat einlesen können, um daraus eine grafische Visualisierung, der für die Aufgabenlösung extrahierten Dateien, anzeigen zu können. Das Editor Modul muss erweiterbar und sprachenunabhängig funktionieren. Hierfür implementiert dieses das Language Server Protocol, um als Client mit verschiedenen Language Servern kommunizieren zu können.
Die vom Language Server empfangenen Daten werden durch den Language Proxy aufbereitet. Basierend auf dem Standard des Language Server Protocol und einer Konfigurationsdatei, werden Informationen entfernt oder hinzugefügt. Um für die Aufgabenlösung freigegebene Dateien zu visualisieren, wird ein File Proxy benötigt. Dieser ermöglicht es, Abschnitte innerhalb der zu bearbeitenden Dateien auszublenden und für die Ausführung der Tests wieder hinzuzufügen.
Der **Workspace Server** beinhaltet das auszulagernde Projekt, die Laufzeitumgebung (*Runtime*) für die Testausführung und einen oder mehrere Language Server. Die entfernte Verwaltung des Projektes ermöglicht es, die Vorteile einer Online-Entwicklungsumgebung zu nutzen und alle ressourcenintensiven Vorgänge auszulagern. Gleichzeitig wird eine klare Abgrenzung zwischen Projekt und Crowd-Entwickler geschaffen.

5.2. ARCHITEKTUR

Abbildung 5.2 zeigt die geplante Architektur des Prototypen. Dabei wurden die in Abschnitt 5.1 erarbeiteten Komponenten eingebettet und deren Kommunikation gekennzeichnet. Die Module der polylith-Entwicklungsumgebung kommunizieren dabei mit dem Workspace-Server über Proxy-Instanzen. Diese Strukturidee folgt dem Software-Architektur Design-Pattern *Proxy-Muster* [Bus98].
Die polylith-Entwicklungsumgebung ist hierbei der Client, der Proxy der Vermittler und ein Workspace die Komponente, welche die Originaldaten integriert. Dieser Aufbau ermöglicht die Aufbereitung von Quelltexten und Antworten vom Language-Server. Gleichzeitig wird der direkte Zugriff auf wichtige Ressourcen verhindert. Weiterhin ermöglicht diese Strukturentscheidung den einfachen Austausch der Proxys, falls Anpassungen durch Abhängigkeiten bei der umzuwandelnden Programmiersprache nötig werden. Dies tritt beispielsweise beim Einlesen der natürlichsprachlichen Tests über den File-Proxy auf.
Im Prototypen werden diese Python-spezifisch mit dem Programm *Behave*[1] umgesetzt. Für andere Programmiersprachen kommen Tools wie Jnario[2], Cucumber[3] oder NatSpec[4] zum Einsatz. Dabei ist die Ausführung der Tests und dessen Ausgabe je nach verwendetem Tool unterschiedlich und muss vom File-Proxy aufbereitet werden.

[1] `http://pythonhosted.org/behave/` (10.6.2017)
[2] `http://jnario.org` (1.7.2017)
[3] `https://cucumber.io` (1.7.2017)
[4] `http://nat-spec.com` (1.7.2017)

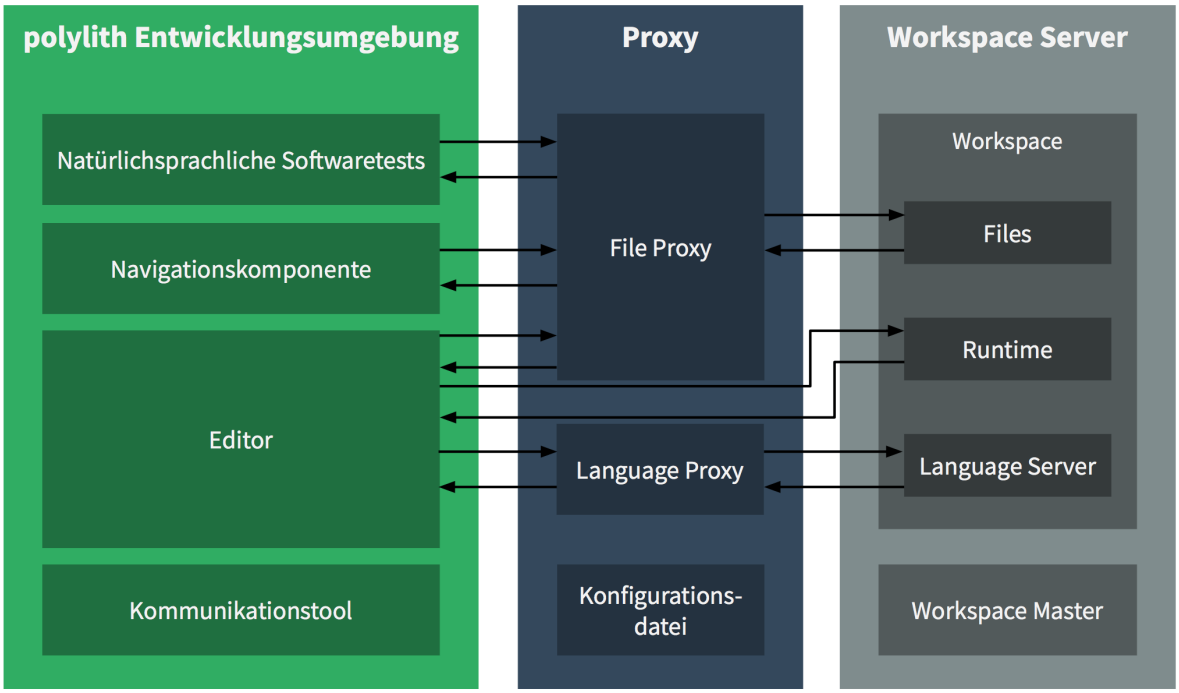


Abbildung 5.2.: Architektur Überblick des Prototypen

Der Language-Proxy ist von der Sprachabhängigkeit weniger betroffen, da er auf Basis des sprachunabhängigen Language-Server-Protokolls arbeitet. Dieses gibt das genaue Datenformat an, welches zwischen dem Editor und Language-Server eines Workspaces verwendet wird. So ist beispielsweise bei der Ausführung der Codevervollständigung, der Rückgabewert immer eine Liste aus *CompletionItems*, welche eine vom Standard vorgegebene Struktur aufweisen müssen. So kann der Language-Proxy die Aufbereitung auf diesen definierten Modellen ausführen. Die Filterung in beiden Proxys wird durch die Konfigurationsdatei gesteuert. Um das Gesamtkonzept skalierbar zu gestalten, müssen die Proxy Komponenten so entworfen werden, dass für jede Aufgabe eine Instanz von ihnen verteilt (*deployt*) werden kann. Hierfür müssen weitere Performance-Untersuchungen durchgeführt werden. Im Rahmen dieser Arbeit liegt der Fokus zunächst auf einer schnellen Umsetzung des Prototypen, um das Gesamtkonzept evaluieren und verbessern zu können.
Weiterhin besitzt der polylith-Editor eine direkte Verbindung zur Runtime eines Workspaces, um die Ausführung von Tests zu ermöglichen. Letzte zu beschreibende Komponente ist der Workspace Master. Dieser ist zuständig für die Verwaltung und das Erstellen einzelner Workspaces. Gleichzeitig kann er später dafür dienen, die Konfigurationsdatei für zugehörige Proxys, durch Analyse des Quelltextes zu erstellen.

5.3. IMPLEMENTIERUNG

Im folgenden Abschnitt wird die prototypische Implementierung des Konzepts beschrieben. Hierfür wurden im Kapitel 3 ähnliche Konzepte und deren technische Umsetzungen betrachtet. Um weiterhin den Stand der Technik für die Implementierung zu nutzen, wurden von bereits existierenden Online-Entwicklungsumgebungen verwendete Technologien analysiert.

5.3.1. WORKSPACE SERVER

Für die Umsetzung des Workspace Servers wird Eclipse Che (Kapitel 3.2.2) verwendet. Die Entscheidung stützt sich auf mehrere Faktoren. Das Projekt deckt die Anforderung ab, verschiedene Projekte und Testumgebungen ausführbar zu machen (Anforderung M40). Hierfür nutzt Eclipse Che die Open-Source-Software Docker, welche die Ausführung von Anwendungen in isolierten Linux Umgebungen (Containern) ermöglicht. Weiterhin bietet Eclipse Che durch den Einsatz von RESTful API's gute Schnittstellen zu seinen Services und Funktionalitäten, um diese in entkoppelten Komponenten ansteuern zu können. Zudem ist das Projekt Open Source und kann somit beliebig erweitert und angepasst werden. Für die Programmiersprachen-Unterstützung verwendet Eclipse Che das Language Server Protocol. Die ausgewählten Language Server werden in angelegte Workspaces eingefügt und können über API's verwendet werden. Der Zugang zum gesamten Softwareprojekt, welchen der Language Server benötigt, ist somit gesichert und muss nicht auf Client Seite installiert werden.

Im Vergleich zu Eclipse Orion (Kapitel 3.2.1) sind die Vorteile deutlich. So bietet die Basisimplementierung von Orion keine Container Unterstützung. Weiterhin wurde auch die *Codenvy*-Plattform in Betracht gezogen. Bei dieser handelt es sich um die kommerzielle Version von Eclipse Che und bietet unter anderem das zusätzliche Feature, Teams anzulegen, um den Zugang zu Workspaces zu steuern. Ein feingranulares Rechtesystem auf Datei-Ebene bietet diese Lösung aber nicht. Zudem ist Codenvy kostenpflichtig und nicht für die kommerzielle Weiterentwicklung vorgesehen.

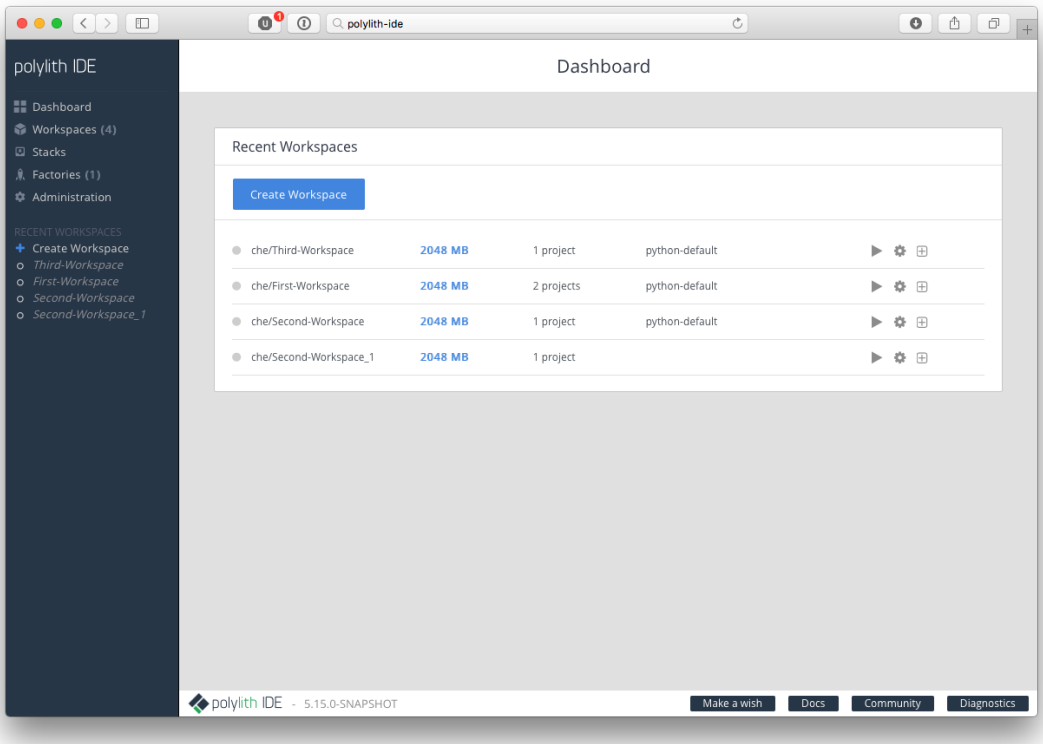


Abbildung 5.3.: Angepasste Eclipse Che Anwendung, zeigt die Verwaltung einzelner Workspaces

Zunächst ist die Nutzung von Eclipse Che ausschließlich als Workspace Server vorgesehen. Die IDE Komponente (Frontend) wird nicht verwendet. Da die polylith-IDE iterativ entwickelt wird und für die crowdbasierte testgetriebene Entwicklung angepasst wird, werden viele Funktionalitäten der Eclipse Che IDE nicht benötigt. Somit müssten aus dem Projekt viele Module entfernt und gleichzeitig ein aufwendiges Rechtesystem hinzugefügt werden.
Die Verwendung der Eclipse Che IDE ist zu einem späteren Zeitpunkt nicht ausgeschlossen. Um ein schnelles Prototyping durchführen zu können, werden unabhängige Services implementiert, welche mit den Eclipse Che RESTful API's kommunizieren. So wird die Einarbeitung in komplexe Frameworks wie dem Google Web Toolkit umgangen und eine frühzeitige Evaluation des Konzepts gesichert. Um Eclipse Che lokal zu verwenden, wird das bearbeitete Repository in einen Docker Container eingebunden und anschließend durch Che bereitgestellte Skripte gestartet.
Im Rahmen der Arbeit wurden Fehler und Implementierungswünsche an die Entwicklung von Eclipse Che über GitHub gestellt[5]. Die Fragen und Anmerkungen wurden immer vom Entwicklerteam beantwortet. Verbesserungsvorschläge wurden angenommen und meist zeitnah umgesetzt. Essentiell für die Erarbeitung des Prototypen war das Einbinden eines neuen Python Language Server in Eclipse Che. Dabei wurde der Language Server von Sourcegraph[6] durch den leistungsfähigeren Palantir-Server[7] ersetzt. Aufgrund häufiger Releases der Che-Anwendung mussten Funktionalitäten des polylith IDE Prototypen angepasst werden. Beispielsweise wurde die Che HTTP API für die Kommunikation mit Language-Server auf Websockets umgestellt[8].
Zusammenfassend kann der Workspace-Server als Konfigurationsbackend des Task-Distributor gesehen werden (Abbildung 5.3). Je nach Programmiersprache und Projektabhängigkeiten wird eine initiale Runtime-Konfiguration vom Nutzer bestimmt und angelegt. Diese beschreibt den auszuführenden Docker-Container. Im Anschluss wird der Projektcode über Git in den Workspace geladen. Durch Eclipse Che werden gleichzeitig APIs für die Kommunikation bereitgestellt. Diese ermöglichen es der polylith-IDE Komponente, Inhalte von Dateien zu bearbeiten, Sprachenunterstützung vom Language-Server zu erhalten und Kommandos innerhalb der Runtime anzustoßen.

5.3.2. POLYLITH-IDE

Um eine schnelle Umsetzung des Prototypen zu erlauben, wurde *Python 3*[9] und das Webframework *Django 1.11*[10] verwendet. Letzteres ermöglicht durch sein ausgereiftes ORM (Objectrelational mapping) und dem Django-Admin eine einfache Möglichkeit, Testdaten anzulegen und zu verwalten. Weiterhin bestehen einschlägige Erfahrungen in den genannten Technologien, was die Umsetzung eines Proof of Concept sichert. In der finalen Plattform wird die von der Django-Anwendung abgedeckte Komponente auf mehrere Services aufgeteilt. So sind ein Workspace-Service, Task-Service und User-Service geplant, welche die Backenddaten verwalten.

[5] `https://github.com/eclipse/che/issues/4553`,`https://github.com/eclipse/che/issues/4555`, `https://github.com/eclipse/che/issues/4976`,`https://github.com/eclipse/che/issues/5011`, `https://github.com/eclipse/che/issues/5011`,`https://github.com/eclipse/che/issues/5389`, `https://github.com/eclipse/che/issues/5407` (1.7.2017)
[6] `https://github.com/sourcegraph/python-langserver` (20.72017)
[7] `https://github.com/palantir/language-servers` (20.7.2017)
[8] `https://github.com/eclipse/che/issues/5075` (1.7.2017)
[9] `https://www.python.org` (30.6.2017)
[10] `https://www.djangoproject.com` (30.6.2017)

In der prototypischen Umsetzung können Aufgaben mit Hilfe des Django-Admins angelegt und gespeichert werden. Um weiterhin einen schnelleren Fortschritt in der prototypischen Umsetzung zu ermöglichen und frühzeitig das Konzept zu evaluieren, werden die Komponenten **polylith Entwicklungsumgebung** und **Proxy** in einer Django Anwendung umgesetzt. Zunächst wurde versucht, die Services wie geplant getrennt zu entwickeln. Dieses Vorgehen brachte einen deutlichen Overhead mit sich. Zu einem späteren Zeitpunkt wird die Funktionalität in eine eigene Anwendung extrahiert.

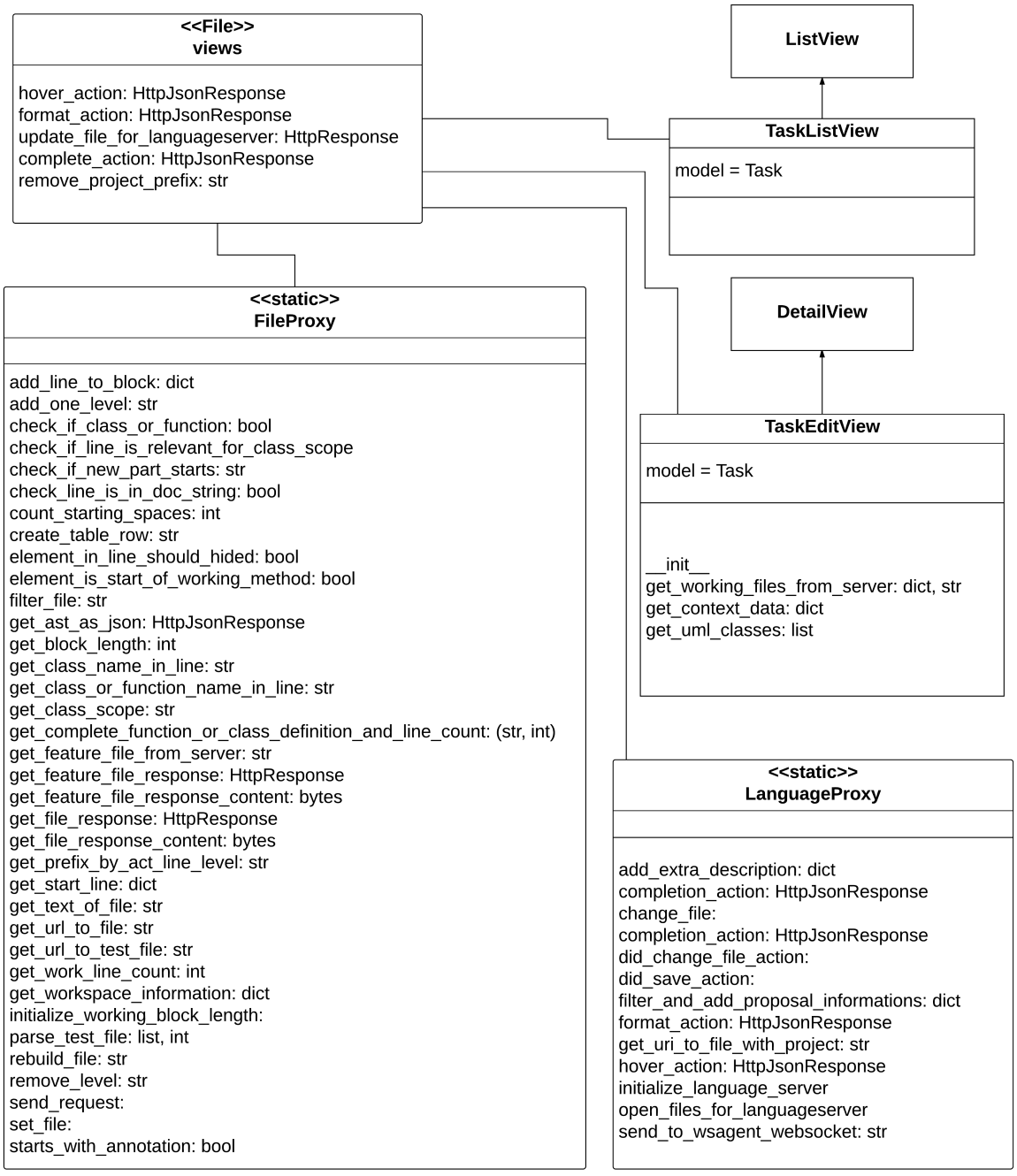


Abbildung 5.4.: Klassen und File Struktur der polylith-IDE

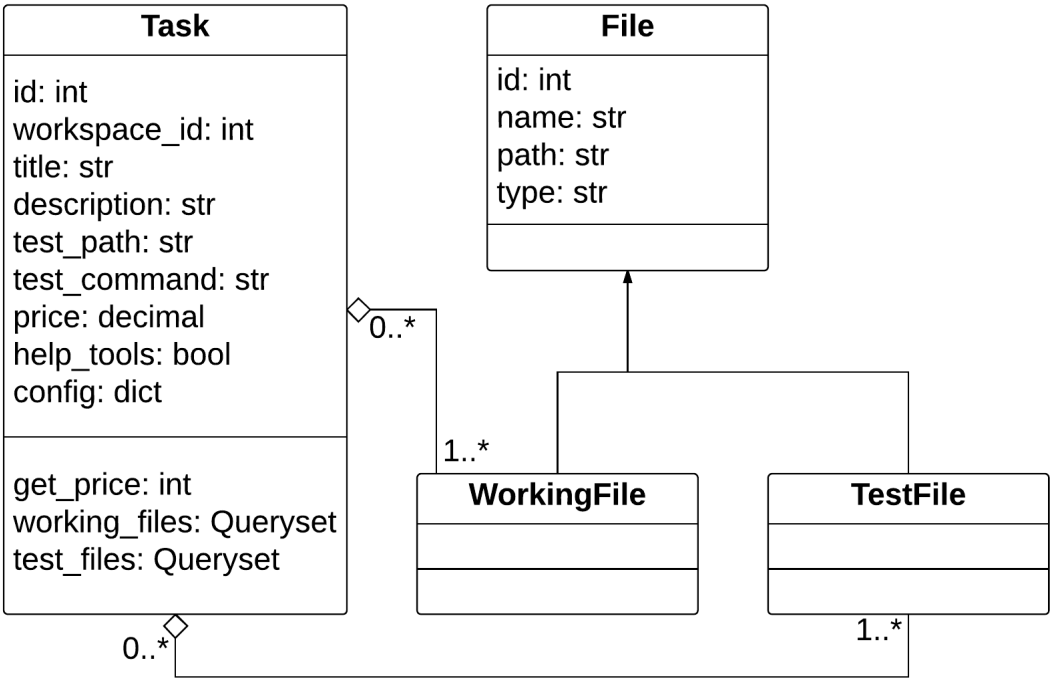


Abbildung 5.5.: Modelle für das Anlegen von Aufgaben.

Bei der Umsetzung wird auf eine möglichst unabhängige Implementierung geachtet, um später die Proxy Komponenten zu extrahieren und in eigenen Services auszuführen. Innerhalb der Django Anwendung wird eine klare Trennung zwischen Frontend- (HTML-Templates, Javascript Dateien) und Backendfunktionalitäten (Empfangen und Senden von Requests) durch Django-Apps geschaffen. Die Frontend Komponenten werden in ein *Bootstrap3*[11] MockUp der Plattform eingebettet. Um die einzelnen Module vom Nutzer anpassbar zu machen, wird weiterhin *jquery UI*[12] verwendet.
Abbildung 5.4 zeigt die Klassen- bzw. Dateistruktur der polylith-IDE. Dabei werden zunächst nur Bestandteile des Backends dargestellt. Beim Aufruf der Anwendung wird ein HTTP-Request von der Django Anwendung entgegengenommen und eine in den `views` definierte Funktion aufgerufen. In diesen werden Methoden aus der `LanguageProxy` oder `FileProxy` Klasse verwendet. Die `LanguageProxy` Klasse beinhaltet Funktionalitäten für die Kommunikation mit dem Language-Server. Um mit dem Language-Server im Eclipse Che Workspace zu kommunizieren, wird eine Websocket-Verbindung aufgebaut. Abbildung 5.6 stellt diese Kommunikation grafisch dar. Das Frontend (*Editor*) sendet dabei zunächst eine Anfrage an das Backend (Abbildung 5.6 **1.**). Die Django Anwendung sendet diese über die Websocket-Verbindung weiter und hört auf die Antwort des Eclipse Che Workspace Agents (Abbildung 5.6 **2.**). Dieser kommuniziert intern mit dem gestarteten Language-Server und sendet eine Nachricht zurück an die polylith-IDE. Die Anfragen werden mit dem JSONRPC-Protokoll gesendet. Empfangene Antworten können mittels einer vorher spezifizierten Konfigurationsdatei gefiltert werden. Die Kommunikation mit der File-API des Eclipse-Che Workspaces funktioniert über HTTP-Requests.

[11] `http://getbootstrap.com` (2.7.2017)
[12] `https://jqueryui.com`(2.7.2017)

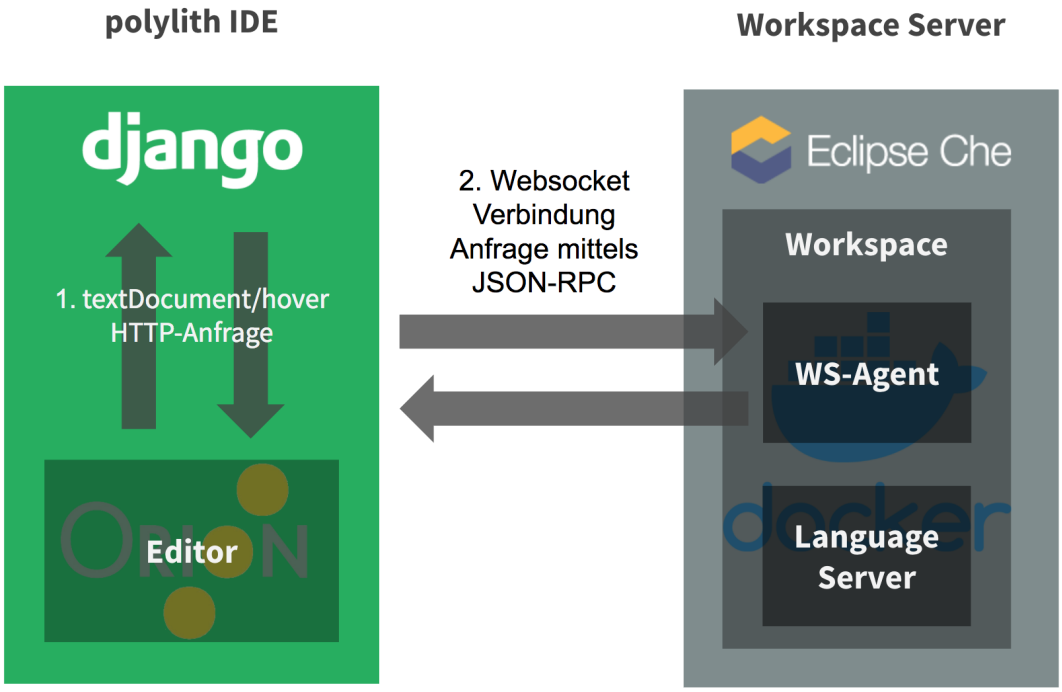


Abbildung 5.6.: Kommunikation zwischen der Django Anwendung und dem Che Workspace Server ohne implementierten Proxy Service

Die Methoden in der `FileProxy` Klasse filtern die erhaltenen Inhalte und fügen diese für das Speichern auf dem Server wieder zusammen.
Um Aufgaben in der polylith-IDE zu erstellen, können über den Django-Admin `Task` Objekte angelegt werden. Die zugehörige Datenstruktur wird in Abbildung 5.5 dargestellt. Eine Aufgabe beinhaltet somit folgende Informationen:

Titel: Titel der Aufgabe.

Beschreibung: Zusätzliche Beschreibung der Aufgabe und des Kontexts.

Workspace ID: Identifier für den Eclipse Che Workspace, in dem die Aufgabe ausgeführt wird.

Projekt-Name: Name für das Projekt innerhalb des Workspaces.

Pfad zur Feature-Datei: Relativer Pfad innerhalb des Projekts zur Datei, welche die natürlichsprachlichen Tests beinhaltet.

Konfiguration: JSON Datei, die die ausgeblendeten Klassen und Funktionen beinhaltet. Weiterhin können Klassen und Methoden mit zusätzlichen Informationen angereichert oder aus der Vorschlagsliste ausgeblendet werden.

Test-Kommando: Befehl, welcher für die Ausführung der Tests aufgerufen werden muss.

Working Files: Dateien, die für den Nutzer sichtbar sein sollen.

Test Files: Dateien, die den Quellcode der Tests beinhalten.

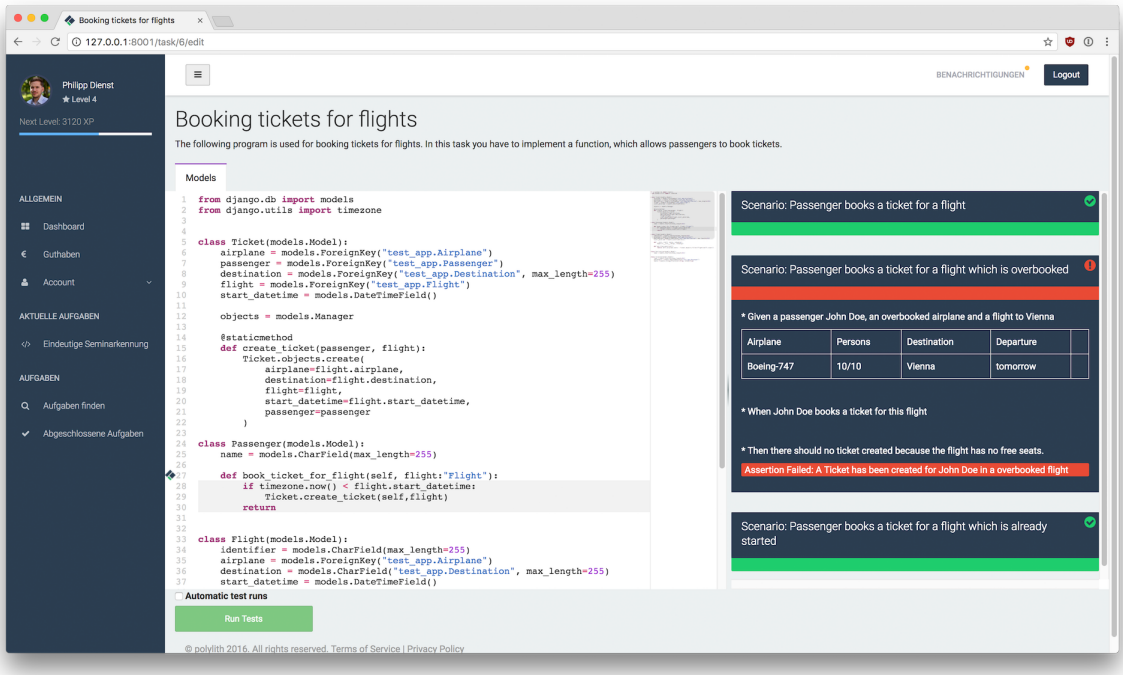


Abbildung 5.7.: Zeigt das polylith-IDE-Frontend. Links die ausgeklappte Navigation, in der Mitte der Editor, rechts die geparsten Tests. In diesem Szenario darf der Nutzer nur die Funktion `book_ticket_for_flight(self, flight)` implementieren (hervorgehoben durch die graue Hintergrundfarbe)

In späteren Iterationen der Anwendung müssen diese Aufgaben automatisch erstellt werden. Für die erste Evaluation der polylith-IDE werden diese manuell angelegt.
Abbildung 5.7 zeigt das umgesetzte Frontend der polylith-IDE. Im Folgenden wird die Umsetzung der einzelnen Bestandteile genauer beschrieben.

NATÜRLICHSPRACHLICHE TESTS

Die natürlichsprachlichen Tests bilden neben der Aufgabenbeschreibung die eigentlichen Spezifikationen für die Aufgabe. Da der Prototyp zunächst mit Python Aufgaben evaluiert werden soll, wird das sprachenspezifische Tool Behave eingesetzt. Für die Visualisierung im Frontend der poylith-IDE werden die Feature Dateien in HTML geparst. Da kein Parser für GherkinSyntax zu HTML gefunden wurde, der Anpassungen, Tabellen-Darstellung und das Parsen einzelner Szenarios und Steps ermöglicht, musste diese Komponente selber implementiert werden. Es handelt sich dabei zwar um *Markdown*-Syntax, trotzdem müssen die einzelnen Backgrounds, Scenarios und Steps zunächst aus der Datei erkannt werden, um diese in HTML angepasst darzustellen. Das spezifische Rendering ermöglicht, nach Ausführung der Tests Fehler an die HTML-Elemente hinzuzufügen und Datentabellen darzustellen (Abbildung 5.8). In weiteren Iterationen der polylith-IDE wird diese Funktionalität an das Modul der polylith-Plattform angepasst, welches ein sprachenunabhängiges Verfassen natürlichsprachlicher Tests ermöglicht. Dieses Modul ist nicht Teil dieser Arbeit.

Um die Testausführung anzustoßen, wird zwischen der polylith-IDE und der *Command API* des Workspaces eine Websocket-Verbindung aufgebaut. Nach dem Versenden eines Befehls wird dieser innerhalb des Workspace-Containers ausgeführt. Das Ergebnis wird über die Websocket-Verbindung, zeilenweise an die polylith-IDE zurück gesendet. Die einzelnen Zeilen werden so analysiert, dass Fehler der Testausführung anschließend in den natürlichsprachlichen Tests visualisiert werden können (Abbildung 5.8 rotes Szenario). Diese Funktionalität ist somit im Prototypen dieser Arbeit ebenfalls sprachenabhängig und muss in weiteren Iterationen für andere Programmiersprachen angepasst werden.

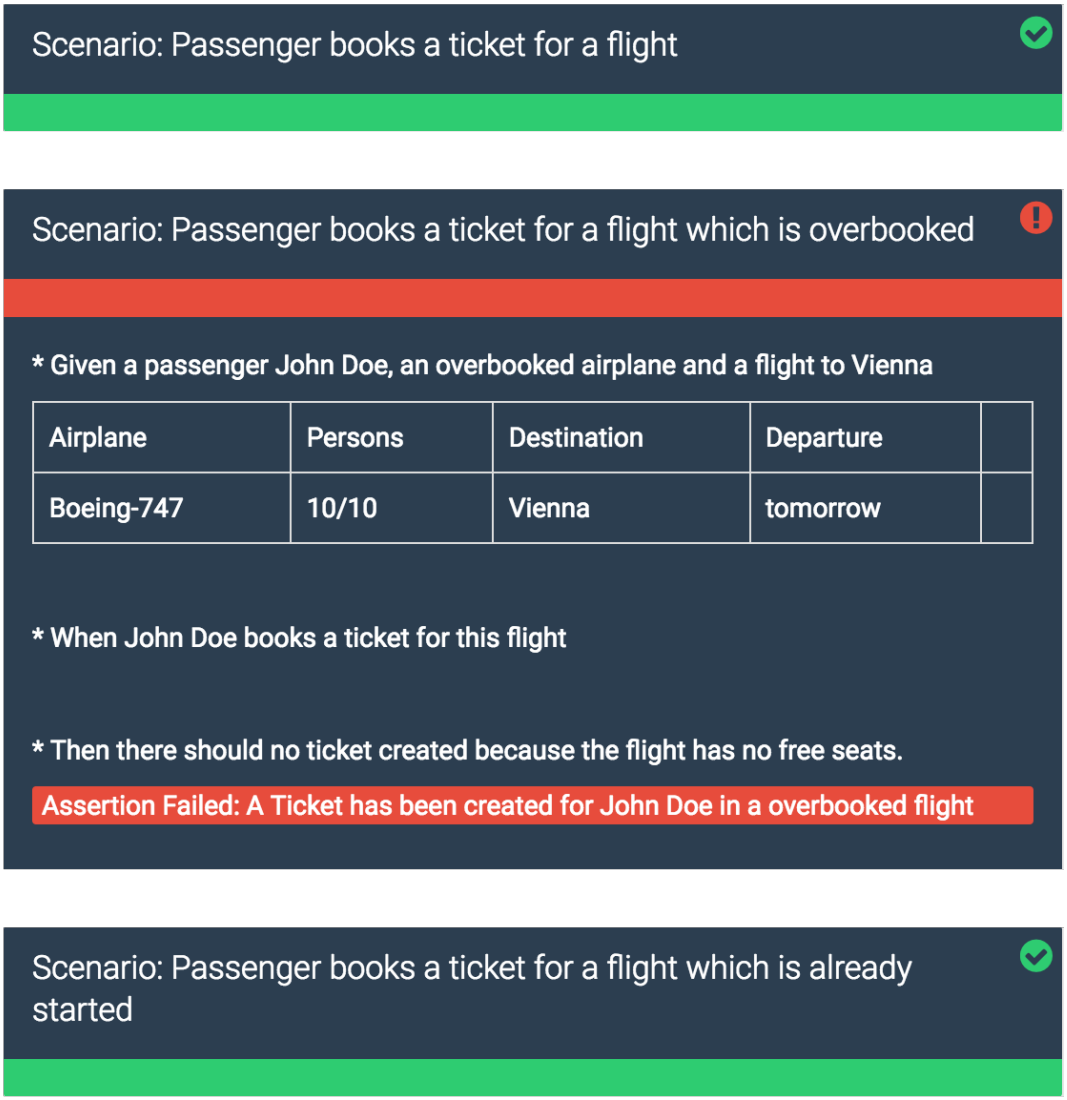


Abbildung 5.8.: Darstellung der natürlichsprachlichen Tests in HTML

EDITOR

Der Editor ist eine Frontendkomponente der polylith-Entwicklungsumgebung. Für die Umsetzung wurden verschiedene, in Javascript implementierte, Texteditoren betrachtet und evaluiert:

CodeMirror[13] ist ein sehr verbreiteter und leichtgewichtiger Editor. Er wird in den Entwicklertools von Firefox, Safari und Chrome verwendet.

Ace[14] wird in der Cloud9 IDE[15]und GitHub verwendet. Die Basisversion bietet Syntax Überprüfungen für HTML, CSS und Javascript. Er besitzt ansonsten ähnliche Eigenschaften wie CodeMirror.

Monaco[16] ist die Web Version des Desktop Editor Visual Studio Code[17]. Die Basisversion bietet Syntax Überprüfungen für HTML, CSS, LESS, SCSS, JSON, TypeScript und Javascript.

Eclipse Orion Code ist der Editor aus dem Eclipse Orion Projekt. Er wird im Eclipse Che Projekt verwendet (detaillierter in Kapitel 3.2.1 vorgestellt).

Alle Editoren erfüllen die in Kapitel 2.3 erarbeiteten Anforderungen (M20, S20 Syntaxhervorhebung). Aufgrund der Entscheidung für Eclipse Che als Workspace-Server wurde für die prototypische Implementierung das Editor Widget von Eclipse Orion gewählt. Die bereits vorhandenen Sprachunterstützungen haben für die Auswahl keine Rolle gespielt, da diese über die Client Implementierung des Language-Server-Protokolls erfolgen. Diese Erweiterung des Widgets konnte sehr gut umgesetzt werden. Um Funktionalitäten wie das Hovern oder Codevorschläge anzupassen, müssen *Services* überschrieben bzw. angepasst werden. Beim Aufruf eines solchen Services wird der *Editor Context* übergeben. Bei diesem handelt es sich um eine Objekt-Referenz, welche eine Zwei-Wege-Kommunikation zwischen Service und dem Orion Editor ermöglicht. Diese Kommunikation funktioniert asynchron und folgt dem *Inversion of Control* (IoC) Design-Prinzip. So kann über das *Editor Context* Objekt der aktuelle Zustand einer Editor Instanz abgefragt werden. Quelltext 1 zeigt wie der Inhalt des Editors angefragt werden kann. Die `getText` Funktion gibt dabei ein *Promise*[18] Objekt zurück. Es repräsentiert einen Proxy für eine Variable, deren Wert bei der Erstellung unbekannt ist und über asynchrone Vorgänge (beispielsweise HTTP-Requests) geladen wird. Dies ermöglicht asynchronen Funktionen in gleicher Weise Werte zurückzugeben wie synchrone Methoden. Ein Promise kann dabei zu einem *fullfilled* mit dem angeforderten Wert oder ein *rejected* mit Grund werden. Über die `then`-Methode (siehe Quelltext 1) sind *Handler* assoziiert, welche nach der Statusänderung des Promise-Objekts aufgerufen werden. Auf diesem Verfahren basieren sämtliche Serviceprovider des Orion Code Edit Widgets. Somit wird es ermöglicht, innerhalb der Services asynchrone HTTP-Anfragen mittels *Ajax* an den Language-Server zu senden. Hierfür wurde die Ajax Implementierung der Javascript-Bibliothek *jQuery*[19] verwendet, welche das Promise Interface für seine Rückgabewerte implementiert hat.

[13] `http://codemirror.net` (1.7.2017)
[14] `https://ace.c9.io` (1.7.2017)
[15] `https://c9.io` (1.7.2017)
[16] `https://microsoft.github.io/monaco-editor/` (1.7.2017)
[17] `https://code.visualstudio.com` (12.8.2017)
[18] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise` (2.7.2017)
[19] `https://jquery.com` (4.7.2017)

```
editorContextObject.getText().then(function(text) {
    console.log(text);
});
```

Quelltext 1: Beispiel für die Anfrage des Editor Inhalts über den *Editor Context*.

```
function hover_command_request(line, character, file_path) {
    var post_data = {
        'csrfmiddlewaretoken': '{{ csrf_token }}',
        'line': line,
        'character': character,
        'file_path': file_path,
    };
    return $.ajax({
        type: "POST",
        url: "{% url "hover_action" object.id %}",
        data: post_data,
        success: function (data) {
        },
        error: function (data) {
        }
    });
}
function process_ajax_request() {
    return hover_command_request(
        act_line, act_char, file_path
    ).then(function (data) {
        var json_data = JSON.parse(data);
        handle_json_data(json_data);
    })
}
```

Quelltext 2: Beispiel für die Verwendung eines Ajax Requests, mittels *Promise* Handlings

Quelltext 2 zeigt, wie der Rückgabewert einer jQuery-Ajax-Anfrage weiterverarbeitet werden kann. Die Anfrage wird dabei zunächst an die Django Backendanwendung geschickt. Diese besitzt eine Websocket Verbindung zum Language-Server, welcher im Eclipse Che Workspace läuft. Abbildung 5.6 stellt diese Verbindungen grafisch dar. In einem nächsten Implementierungsschritt muss die Funktionalität in eine Proxy-Anwendung (wie in Abschnitt 5.2 beschrieben) extrahiert werden.
Zunächst wurde der *HoverProvider* und der *ContentAssistProvider* des Orion Code Editors um die beschriebene Ajax Kommunikation erweitert. Zusätzlich wurde ein eigenes Kommando für die automatische Formatierung hinzugefügt, welches in den *COULD*-Anforderungen gefordert ist. Weiterhin sendet der *InputEventListener* den aktuell angezeigten Inhalt an die File-API des Workspaces, um die Dateien zu aktualisieren und persistieren.
Um dem Nutzer mehrere Dateien zugänglich zu machen, werden alle zuvor angegebenen Files im Javascript Code zwischengespeichert. Über eine Tab-Leiste oberhalb des Editors, kann zwischen diesen umgeschaltet werden. Eine solche Funktionalität liefert das Orion Code Edit Widget nicht von Grund auf und musste deshalb selbst implementiert werden.

Weiterhin wurde der Orion-Editor um eine eigene Validierung erweitert, um nur an einer Position der Datei die Bearbeitung des Nutzers zuzulassen. Dabei überprüft das Widget bei jeder Änderung, ob sich die aktuelle Eingabe des Nutzers innerhalb der freigegebenen Spanne befindet. Fügt der Nutzer weiteren Code oder eine neue Zeile am Ende dieser Spanne ein, wird dies vom Editor erkannt und der Bereich vergrößert. Visualisiert wird diese Spanne durch eine graue Hintergrundfarbe der Codezeile (Abbildung 5.9). Außerhalb dieses Bereichs darf der Nutzer keine Änderungen tätigen. Beim Versuch springt der Cursor in den Bereich, welchen er bearbeiten darf.

```
55      @property
56      def is_cancelled(self):
57          return bool(self.cancellation_date)
58
59      @property
60      def is_cancelable(self):
61          if self.is_cancelled:
62              return False
63          return self.state == choices.SEMINAR_PARTICIPATION_APPROVED
64
```

Abbildung 5.9.: Der Bereich, in dem der Nutzer Änderungen vornehmen darf. Das Symbol links verdeutlicht, um welche Funktion es sich handelt.

KONFIGURATIONSDATEI

Die Konfigurationsdatei ist für die Filterung und Anreicherung des Quellcodes zuständig und wird von den Proxy Komponenten der polylith-IDE verwendet. Diese wird zunächst als JSON-Datei manuell spezifiziert (Quelltext 3). In späteren Iterationen muss diese beim Anlegen der natürlichsprachlichen Tests spezifiziert werden, damit Task-Distributoren vor der Aufgabenpublikation weitere Funktionen manuell ausblenden können. Hierfür muss die Erstellung in das geplante Tool eingebunden werden, welches ein sprachenunabhängiges Verfassen natürlichsprachlicher Softwaretests ermöglicht. Da dieses Tool zum aktuellen Entwicklungszeitpunkt nicht vorhanden ist, wird in dieser Arbeit Fokus auf die Struktur der Konfigurationsdatei gelegt. Um genau zu spezifizieren welche Elemente nicht sichtbar sein sollen, benötigen diese eine eineindeutige Kennung. Hierfür muss jede Funktion bei der Analyse des Quellcodes eine solche Kennung erhalten. Anschließend müssen diese in die Konfigurationsdatei eingetragen werden. Da die erste prototypische Umsetzung diese Test-Analyse nicht beinhaltet und die Elemente des Quellcodes zunächst keine eineindeutige Kennung zugewiesen bekommen, wird eine einfachere Identifizierung gewählt. Da der Prototyp zunächst Python Quellcode filtern muss, werden Funktionen und Methoden über den *Fully Qualified Name* spezifiziert. Dieser bildet sich aus dem relativen Pfad innerhalb des zugehörigen Projekts und dem *Qualified Name*[20] der Funktion oder Klasse. Um ein Element auszublenden, wird der entsprechende Identifikator in eine Liste eingetragen (invisible_elements). Weiterhin muss die Methode spezifiziert werden, welche vom Crowd-Entwickler implementiert werden soll (**working_methods**). Diese wird ebenfalls in eine Liste eingetragen, um später auch mehrere Methoden zugänglich zu machen. Neben dem Ausblenden von Funktionen muss die Konfigurationsdatei auch das Anreichern von Informationen ermöglichen. Hierfür können in das **context_informations** *Dictionary* Elemente eingetragen werden.

[20] https://www.python.org/dev/peps/pep-3155/ (9.8.2017)

```
{
  "working_methods": [
    "models.Flight.create_airplane"
  ],
  "invisible_elements": [
    "models.Aiplane"
  ],
  "context_informations": {
    "add": {
      "Ticket": "Ticket for a flight",
      "Destination": "Destination of a flight"
    },
    "remove": [
      "Passenger"
    ]
  }
}
```

Quelltext 3: Beispiel einer Konfigurationsdatei, um Funktionen und Klassen auszublenden und zusätzliche Informationen hinzuzufügen.

Im Unterschied zu den vorhergehenden Spezifikationen, werden in das *Dictionary* **add** und in die Liste **remove** nur die Namen der anzureichernden Elemente eingetragen. Damit sind die Identifikatoren nicht eindeutig. Das liegt an den, vom Language Server Protokoll definierten *Response-Elementen*. Für eine *Hover-Tooltip*-Anfrage sendet der Editor beispielsweise nur die aktuelle Position des Cursors an den Language-Server. Für die Identifikation werden nur der Name und der Methodenrumpf vom Server zurückgesendet. Für eine eineindeutige Zuordnung muss das Language-Server-Protokoll angepasst werden, oder der Editor die Position auswerten, um daraus die Identifikation zu sichern.

LANGUAGE-PROXY

Der Language-Proxy ist in der Umsetzung des Prototypen Teil der Django-Applikation. Seine Aufgabe ist die Kommunikation mit dem Language-Server und die Filterung bzw. Anreicherung der ausgetauschten Daten. Dabei wird die Aufbereitung auf Grundlage, der im Language-Server-Protokolls definierten Datenstrukturen ausgeführt. Ein Beispiel für die Kommunikation mit dem Eclipse Che Workspace wurde in Abschnitt 5.3.2 gegeben.
Die `LanguageProxy` Klasse beinhaltet die Funktionalitäten für die Kommunikation mit dem Language-Server (Abbildung 5.4). Zunächst wird ein HTTP-Request vom Editor an die Django Anwendung gesendet. Die angesteuerte *View*[21] verwendet Methoden aus der `LanguageProxy` Klasse. Für die Anreicherung und Entfernung von *Autocompletion-Items* und *Hover-Tooltips* wird in der Konfigurationsdatei nach den entsprechenden Namen gesucht.

[21] `https://docs.djangoproject.com/en/1.11/topics/http/views/` (9.8.2017)

Für die Filterung werden die Labels der empfangenen *CompletionItems*[22] mit den Elementen der **remove** Liste verglichen und gegebenenfalls entfernt. Bei der Anreicherung der *Hover*-Methode[23] wird im zurückgegebenen *MarkedString* nach dem Namen der Funktion gesucht. Anschließend wird überprüft, ob sich dieser in der Konfigurationsdatei befindet und gegebenenfalls Inhalte hinzugefügt. Anschließend wird das gefilterte Ergebnis an den Editor zurückgesendet und visuell aufbereitet (Abbildung 5.10 und 5.11).

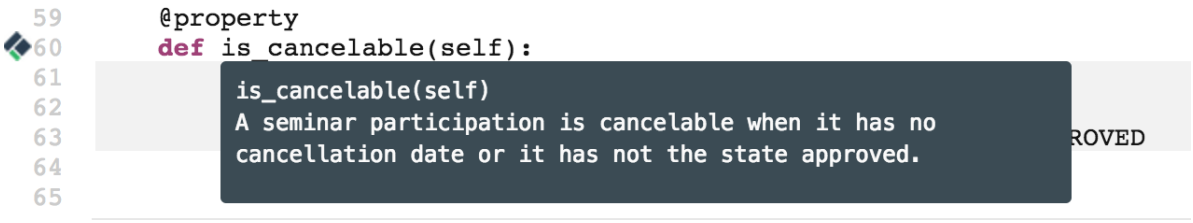


Abbildung 5.10.: Hover-Tooltip Unterstützung. Die zusätzlichen Informationen stammen aus der Konfigurationsdatei.

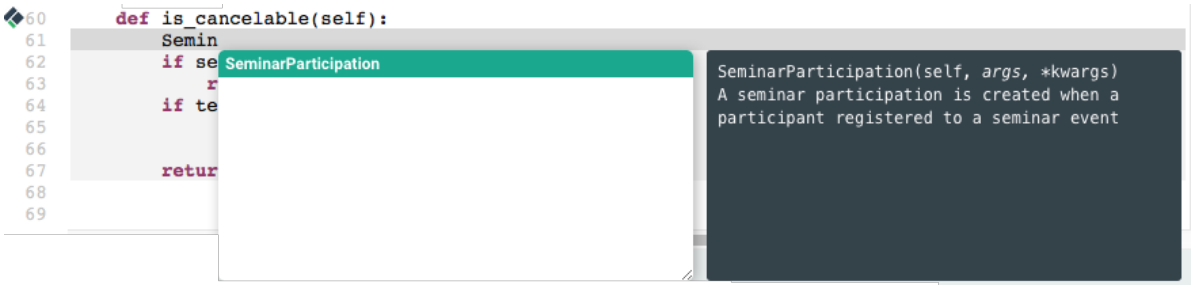


Abbildung 5.11.: Die Autovervollständigung zeigt zusätzlich Informationen aus den Hover-Tooltips an.

FILE-PROXY

Der File-Proxy ist für die Aufbereitung des dargestellten Quellcodes zuständig. Hierfür kommuniziert dieser mit der von Eclipse-Che im Workspace bereitgestellten File-API. Da der CrowdEntwickler nur gewisse Teile des Quellcodes einsehen darf, werden die freigegebenen Dateien zusätzlich gefiltert. Dabei wird der Quelltext zeilenweise abgearbeitet und in Blöcke aufgeteilt. Die Datei wird hierfür als Liste zwischengespeichert. Beinhalten Zeilen eine Klasse oder Funktion, welche für den Crowd-Entwickler nicht sichtbar sein soll, wird statt des Quelltextes ein Platzhalter in die Liste eingefügt. Der ausgeblendete Code wird gleichzeitig in einem *Dictionary* zwischengespeichert. Weiterhin werden die freigegebenen Dateien nach der zu implementierenden Funktion durchsucht. Beinhaltet die Zeile diese, wird ebenfalls ein Platzhalter eingefügt. Anschließend werden alle Zeilen übersprungen, die sich innerhalb des Elements befinden. Dies kann in Python über die Anzahl der Leerzeichen am Anfang einer Zeile berechnet werden.

[22] `https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md#textDocument_completion` (9.8.2017)

[23] `https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md#textDocument_hover`(9.8.2017)

Um den Bereich für die Implementierung im Editor zu spezifizieren, wird die Startzeile und Endzeile berechnet. Der Start ist immer unterhalb der angegebenen Funktionsdefinition. Die Endzeile berechnet sich aus den schon implementierten Zeilen des Crowd-Entwicklers und der Länge der Funktionsdefinition (diese kann in Python über mehrere Zeilen deklariert werden). Die ermittelte Spanne wird anschließend dem Editor Widget übergeben.
Bei der Änderung einer Datei wird der an den Worspace-Server zu sendende Quelltext mithilfe der zuvor zwischengespeicherten Liste zusammengesetzt. Hierfür werden die eingefügten Platzhalter mit dem Quelltext aus dem zwischengespeicherten *Dictionary* ersetzt. Handelt es sich dabei um den Platzhalter für die zu bearbeitende Funktion, wird die empfangene Änderung eingefügt. Durch dieses Vorgehen wird verhindert, dass Änderungen anderer Funktionen an den Server gesendet und gespeichert werden.

5.4. FAZIT

Der im Rahmen dieser Arbeit umgesetzte Prototyp beinhaltet den Großteil der benötigten Komponenten. Nur die Navigationskomponente und das Kommunikationstool wurden noch nicht umgesetzt. Ersteres ist vor allem für die weitere Entwicklung wichtig und sollte in der nächsten Iteration umgesetzt und evaluiert werden. Für die Umsetzung bietet sich die Javascript Bibliothek *JointJs*[24] an. Diese ermöglicht die Visualisierung unterschiedlichster Diagrammtypen mit Javascript.

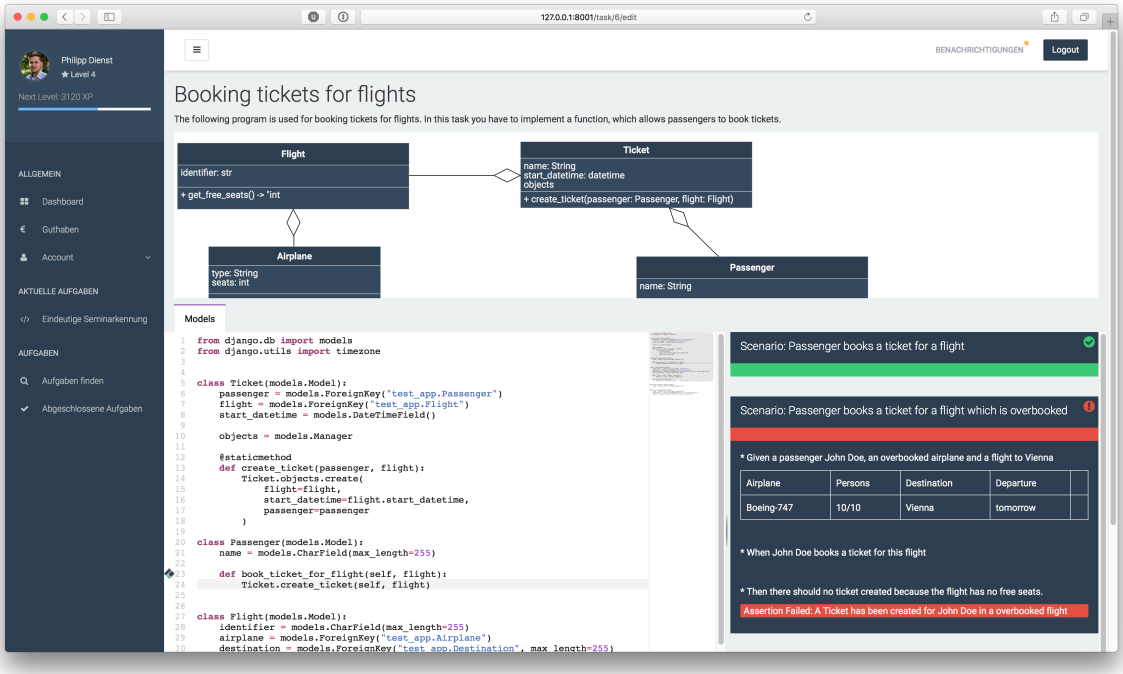


Abbildung 5.12.: Prototyp mit Navigationskomponente

[24] https://www.jointjs.com (10.8.2017)

Abbildung 5.12 zeigt eine erste Umsetzung der Navigationskomponente. Da noch keine automatische Datengenerierung für die UML-Diagramme existiert und die Anbindung an den Editor zeitaufwändig ist, wird die Komponente in der Evaluation nicht verwendet. Damit unterstützt der Prototyp zunächst nicht das Top-Down Vorgehen der Entwickler. Weiterhin wurden bei der Entwicklung des Prototypen sicherheitstechnische Probleme nicht ausreichend betrachtet. So sind die API's von Eclipse Che ungesichert und ohne Autorisierungskontrollen. Crowdentwickler wären in der Lage, zusätzliche Quellcodeabschnitte zu erhalten. Im Produktivbetrieb müssen die einzelnen Komponenten gesichert untereinander kommunizieren. Schnittstellen der Eclipse Che Anwendung dürfen nur für die polylith-IDE Proxys zugänglich sein. Weiterhin muss die Kommunikation und Auswertung der Testausführung über die Proxy-Komponenten umgesetzt werden. Im Prototypen kommuniziert der Editor direkt mit der Eclipse Che Anwendung und stellt somit eine sicherheitskritische Verbindung dar (Abbildung 5.2). Für die Evaluation des Prototypen sind diese ausstehenden Implementierungen unkritisch und nicht relevant.
Abschließend werden die einzelnen Komponenten und deren Entwicklungsstände tabellarisch zusammengefasst (Tabelle 5.1 und 5.2).

Komponente	Stand der Entwicklung	Nächste Schritte
polylith-IDE		
Natürlichsprachliche Softwaretests	Die natürlichsprachlichen Softwaretests werden eingelesen und anschließend für das Frontend in HTML gerendert. Im aktuellen Stand des Prototypen ist diese Komponente auf Feature Dateien (Gherkin Syntax) begrenzt.	Im nächsten Schritt der Entwicklung wird diese Komponente wahrscheinlich eine andere Syntax einlesen müssen. Für die polylith-Plattform wird ein eigenes Tool geschrieben, um sprachenunabhängig natürlichsprachliche Sätze auf Quellcode zu mappen. Dies wird über die Zuordnung auf AST-Knoten ermöglicht. In einer weiteren Iteration muss die Komponente der polylith-IDE an die Ausgabe angepasst werden.
Navigations-komponente	Die Navigationskomponente wurde im ersten Prototypen nicht komplett umgesetzt und wird in der Evaluation nicht verwendet. Bei der Recherche nach einer geeigneten Bibliothek für die Umsetzung wurde JointJs gefunden. Die Bibliothek ermöglicht die Visualisierung verschiedener Diagramme mit Javascript und HTML	In den nächsten Iterationen sollte mit den Funktionalitäten von JoinJs exploriert werden. Dabei sollten verschiedene Diagrammtypen umgesetzt und evaluiert werden. Weiterhin sollte ein Konzept für die Einstellung des Abstraktionsgrad der Grafik erarbeitet werden.
Editor	Der Editor besitzt nach der Implementierung den weitesten Stand und wird in der Evaluation ausführlich getestet. Durch die Verwendung des Language-Server-Protokolls sind die implementierten Funktionalitäten sprachenunabhängig.	In der nächsten Iteration müssen Funktionen wie *Refactoring* und *Go-To-Deklaration* implementiert werden. Weiterhin sollte auf das Feedback der Probanden eingegangen werden und gegebenenfalls die Anforderungen erweitert werden.

Kommunikations-tool	Das Kommunikationstool wurde im ersten Prototypen nicht umgesetzt.	Für die erste Iteration ist ein Hilfe-Button geplant, über den der Crowdentwickler dem Task-Distributor eine Nachricht zukommen lassen kann. Dabei erfährt der Task-Distributor bei welcher Aufgabe sich der Crowdentwickler befindet und kann den aktuellen Stand der Bearbeitung über eine eigene Anwendung (Eclipse Che) einsehen. Um dem Crowdentwickler antworten zu können, muss Eclipse Che um ein Plugin erweitert werden, welches den Versand einer Nachricht zur polylith-IDE ermöglicht.

Tabelle 5.1.: Abdeckung der technischen Anforderungen der polylith-IDE.

Komponente	Stand der Entwicklung	Nächste Schritte
Filterung		
File Proxy	Die geforderten Funktionalitäten des File Proxys wurden vollständig implementiert und können in der Evaluation getestet werden. Es wird das Ausblenden von Funktionen und Klassen in Python ermöglicht. Diese Funktionalität basiert auf Textbasis.	Im nächsten Schritt der Entwicklung wird die Filterung der Datei auf Basis des AST umgestellt. Hierfür wird das Tool *GumTree*[FMB+14] verwendet, welches für verschiedenste Sprachen den AST aus einer Datei als JSON generiert. Über das zurückgegebene Datenformat können Länge und Position einer Funktion oder Klasse ausgelesen werden. Diese Funktionalität soll das textbasierte Verfahren ersetzen.
Language Proxy	Die geforderten Funktionalitäten des Language Proxys wurden vollständig implementiert und können in der Evaluation getestet werden. Die zurückgegebenen Daten des Language-Servers sind dabei nicht so detailliert wie gewünscht. Im aktuellen Stand des Prototypen können Funktionen und Klassen deshalb nur über deren Namen identifiziert werden. Dieser ist in einem Projekt nicht immer ausreichend für die Identifizierung.	Die genannten Probleme lassen sich nur durch eine Anpassung/Erweiterung des Language-Server-Protokolls oder einer gesonderten Implementierung im Language-Proxy lösen. Hierfür müsste der Language-Server den *Qualified Name* des Elements zurückgeben oder der Language Proxy muss diesen selber berechnen.
Workspace Server		
Projekt und Dateien	Projekte und Dateien werden in Workspaces von Eclipse Che verwaltet.	Für die nächste Implementierungsphase muss eine Komponente implementiert werden, die aus den gegeben Quellcode und Tests eine Konfigurationsdatei erstellt.

Language Server	Language Server werden von Eclipse Che in den Workspaces gestartet. Die verfügbaren Language Server werden stetig erweitert. Zum aktuellen Zeitpunkt bietet Eclipse Che eine Unterstützung für PHP, TypeScript, C#, JSON und Python.	Im nächsten Schritt sollten zusätzliche Language Server in Verbindung mit der polylith-IDE evaluiert werden.
Laufzeitumgebung	Der Quellcode und Tests werden in Workspaces von Eclipse Che ausgeführt. Hierfür werden die Projekte und Dateien in Docker Container eingebunden.	Für die weitere Entwicklung sollten die in Che vorhandenen Dockerfiles angepasst werden, um nur benötigte Abhängigkeiten zu installieren und weitere Frameworks zu unterstützen. Hierfür bietet Eclipse Che die Definition individueller *Stacks* an.

Tabelle 5.2.: Abdeckung der technischen Anforderungen der Filterungskomponente und des Workspace Servers.

6. EVALUATION

Im folgenden Abschnitt wird die Evaluation des umgesetzten Prototypen beschrieben. Dabei wird zunächst die geplante Durchführung und die Aufbereitung der Aufgaben vorgestellt. Im Anschluss werden die Ergebnisse ausgewertet und die daraus resultierenden Schlüsse beschrieben. Um den Suchaufwand nach geeigneten Probanden zu minimieren, wurde die Nutzerevaluation mit der Evaluation des Gamification-Konzeptes zusammengelegt. Anforderung für die Teilnahme waren mindestens rudimentäre Programmierkenntnisse in Python.

6.1. VERSUCHSABLAUF

Da Teile der Anforderungen aus Kapitel 2.3 (S10, S30) von menschlichen Faktoren abhängen, ist eine Evaluation mit Probanden erforderlich. Hierfür wird der erarbeitete Prototyp in einer formativen Nutzerevaluation evaluiert. Ziel der Evaluation ist die Kontrolle der technischen, als auch nicht-technischen Anforderungen (Tabelle 2.1). Hierfür erhält der Proband drei Python Programmieraufgaben, welche in unterschiedlichen Editoren gelöst werden sollen. Der Vergleich mit anderen Codeeditoren soll die Verbesserung des Programmverstehens durch das Konzept dieser Arbeit verdeutlichen. Hierfür wurde folgende Konstellation und Reihenfolge gewählt:

Pycharm ist eine etablierte Python-Entwicklungsumgebung von Jet Brains[1]. Für die Evaluation wird die *Professional* Version verwendet, welche den kompletten Funktionsumfang beinhaltet. Diese liefert zum Beispiel Autovervollständigung, Syntaxhervorhebungen und Refactoring für Python. Weiterhin bietet Pycharm Framework-spezifische Hilfen an (Django). Die Entwicklungsumgebung wurde als Vergleich ausgewählt, um die Nützlichkeit des eigenen Konzepts deutlich zu machen.

polylith IDE ohne Hilfen (Editor) beinhaltet den erarbeiteten Editor ohne das Ausblenden von Klassen und Funktionen innerhalb einer Datei. Somit können im gesamten Quelltext Änderungen vorgenommen werden und die zu bearbeitende Funktion wird nicht hervorgehoben. Weiterhin können keine zusätzlichen kontextbezogenen Informationen zu den Hover-Tooltips oder Autovervollständigungsvorschlägen hinzugefügt werden. Auch das Filtern der Vorschläge wird nicht angewendet. Dies entspricht der Umsetzung der *Must*-Anforderungen.

[1] `https://www.jetbrains.com/pycharm/` (11.8.2017)

polylith IDE mit Hilfen (Editor$^+$) beinhaltet den erarbeiteten Editor mit Ausblenden von Klassen und Funktionen innerhalb einer Datei. Weiterhin können zusätzliche kontextbezogene Informationen zu den Hover-Tooltips oder Autovervollständigungsvorschlägen über eine Konfigurationsdatei hinzugefügt werden. Auch ist Ausblenden von Vorschlägen möglich, um beispielsweise Framework spezifische Funktionen zu entfernen. Dieser Editor soll die *Should*-Kriterien überprüfen (S10 und S20).

Dabei muss der Proband für jede Aufgabe einen anderen Editor verwenden. Für die Lösung einer Aufgabe in Pycharm wird dem Probanden eine *Readme*-Datei bereitgestellt, welche die Aufgabenbeschreibung beinhaltet. Dies soll den Ablauf der Aufgabenbearbeitung im Kontext einer Crowdsourcing-Plattform ohne integrierte Entwicklungsumgebung emulieren. Folgende Programmieraufgaben wurden für die Evaluation erarbeitet:

Aufgabe 1: FizzBuzz ist eine Programmieraufgabe, welche in Bewerbungsgesprächen verwendet wird[2]. Es soll eine Funktion implementiert werden, welche eine Zahl zurückgibt. Ist die Zahl durch drei teilbar soll "fizz" zurückgegeben werden. Ist die Zahl durch fünf teilbar, soll "buzz" zurückgegeben werden. Wenn die Zahl sowohl durch drei, als auch durch fünf teilbar ist, soll "fizzbuzz" ausgegeben werden. Die Aufgabe wurde gewählt, um eine Framework unabhängige und bewährte Programmieraufgabe evaluieren zu können. Weiterhin soll überprüft werden, ob das entwickelte Konzept auch auf sehr kleine Softwareprojekte ohne speziellen Kontext anwendbar ist.

Projektstatistiken:
- 9 Dateien insgesamt
- 3 Python Dateien (2 für die Implementierung der Tests)
- 34 Zeilen Quellcode

Aufgabe 2: Booking tickets for flights ist eine Aufgabe, in der eine Funktion implementiert werden soll, welche Instanzen von Tickets erstellt. Dabei darf der zugehörige Flug nicht abgeflogen oder überbucht sein. Überbucht ist der Flug, wenn keine freien Sitzplätze im Flugzeug vorhanden sind. Das Grundgerüst der Aufgabe ist in Python und Django implementiert. Bei der Umsetzung wurde darauf geachtet, dass Probanden ohne Framework spezifisches Wissen die Aufgabe lösen können. Gewählt wurde die Programmieraufgabe, um Probanden innerhalb eines kleineren Projektes evaluieren zu können.

Projektstatistiken:
- 51 Dateien insgesamt
- 26 Python Dateien
- 665 Zeilen Quellcode

Aufgabe 3: Semiplan ist ein größeres Softwareprojekt für die Verwaltung und Planung von Seminaren. Für die Lösung der Aufgabe soll der Proband eine Funktion implementieren, welche prüft, ob eine Seminarteilnahme noch abgesagt werden kann. Dies ist der Fall, wenn die Seminarteilnahme noch nicht abgesagt wurde und der aktuelle Status der Seminarteilnahme "bestätigt" ist. Das Projekt wurde ausgewählt, um die polylith-IDE in einem Kontext von größeren Softwareprojekten zu evaluieren.

Projektstatistiken:
- 1767 Dateien insgesamt
- 335 Python Dateien
- 24684 Zeilen Quellcode

[2] `https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/` (12.8.2017)

Um die Belastung des Probanden bei der Lösung einer Aufgabe mit Hilfe eines bestimmten Editors vergleichbar zu machen, wird der *NASA Task Load Index* (TLX) Fragebogen verwendet [HS88]. Dabei wird die verkürzte Version des NASA-TLX eingesetzt, der keine Fragen für die Gewichtung der Skalen beinhaltet (Abbildung A.1). Diese Entscheidung stützt sich auf kritische Aussagen zur Originalversion [Gro04]. Weiterhin wird die Dauer der zusammengelegten Evaluation verkürzt. Da die geplante Anzahl der Probanden weniger als zehn beträgt, sind die Ergebnisse nicht für eine statistische Analyse geeignet und können somit nur eine Tendenz wiedergeben.
Die Probanden müssen nach jeder gelösten Aufgabe den TLX Fragebogen ausfüllen. Nach der letzten Programmieraufgabe muss ein weiterer Fragebogen ausgefüllt werden. Dieser beinhaltet neben der Einschätzung des polylith-IDE Editors (Abbildung **??**), eine Befragung zum Gamification-Konzept (Abbildung A.3 - A.14). Hierfür wurde der Editor um eine Entwicklerbewertungsansicht erweitert, welche nach Abschluss der Aufgabe visualisiert wird (Abbildung 6.1). Da das Ausblenden von Quellcode für den Probanden nicht ersichtlich ist, wird vor der Ausgabe des Fragebogens die Aufgabe ohne Filterung gezeigt. Somit kann das Feature in die Einschätzung mit einfließen.

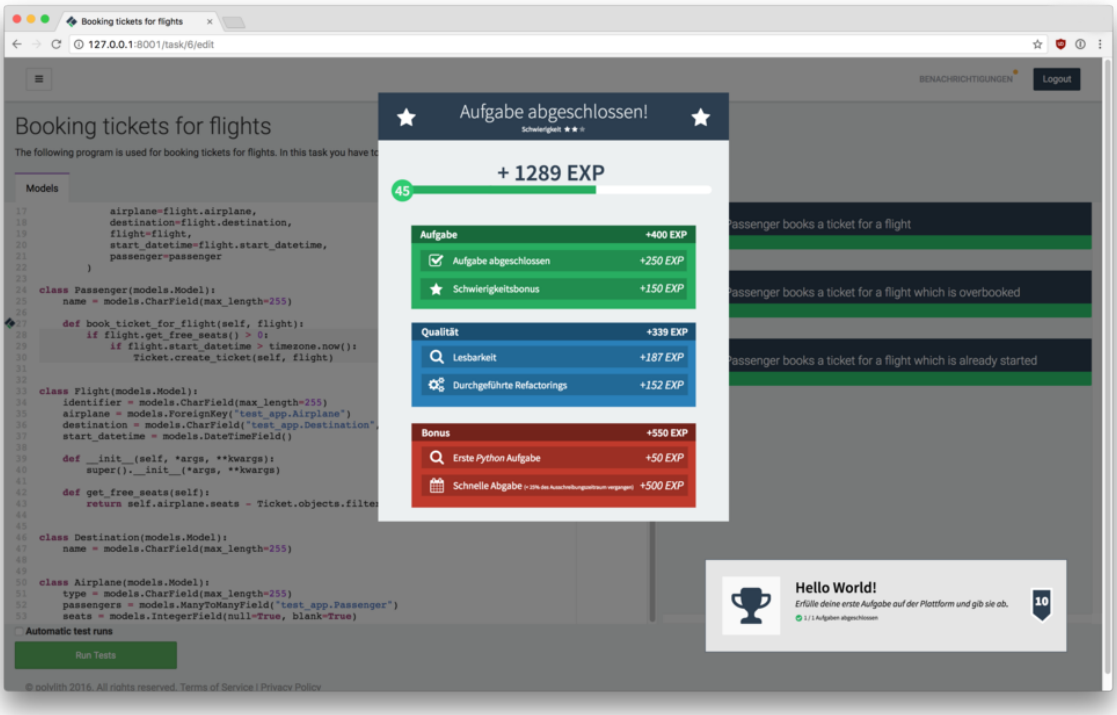


Abbildung 6.1.: Entwicklerbewertungsansicht in der polylith-IDE nach Aufgabenerfüllung

6.2. AUFGABENERSTELLUNG

Für die Evaluation des Projekts werden zunächst Beispielaufgaben in Python an die Probanden ausgegeben. Da zum aktuellen Zeitpunkt ein Tool für sprachenunabhängiges Verfassen von natürlichsprachlichen Tests fehlt, werden diese mit Behave implementiert und ausgeführt.

Es handelt sich dabei um ein Python spezifisches Tool, um Behavior-Driven-Development zu unterstützen. Neben dem Codegrundgerüst müssen zusätzlich Feature-Dateien geschrieben werden, welche die natürlichsprachlichen Beschreibungen der Test-Szenarien beinhalten. Quelltext 4 zeigt ein Beispiel für ein Feature in Behave. Die verwendete Syntax ist *Gherkin*[3] und stammt aus dem Behavior-Driven-Development-Werkzeug *Cucumber*[4]. Das dargestellte Testszenario folgt dem *Given-When-Then* Prinzip. Nach dem Verfassen der Features müssen die einzelnen Sätze mit Funktionen (*Steps*) verknüpft werden (Quelltext 5).

```
Feature: Passengers and airplanes

  Scenario: Passenger goes into airplane
    Given a passenger "John Doe" and a airplane from type "Boeing-747"
    When "John Doe" moves into given airplane
    Then "John Doe" is in airplane "Boeing-747"
```

Quelltext 4: Beispiel für eine Feature-Datei in Behave mit Gherkin Syntax (Given-When-Then)

```
@given('Given a passenger "{}" and a airplane from type "{}"')
def given(name: "str", type_name: "str") -> "models.Airplane":
    passenger = models.Passenger.objects.create(name=name)
    airplane = models.Airplane.objects.create(type=type_name)
    return airplane


@when('"{}" moves into given airplane')
def when(name: "str", airplane: "models.Airplane"):
    passenger = models.Passenger.objects.filter(name=name).first()
    airplane.passengers.add(passenger)


@then('"{}" is in airplane "{}"')
def then(name: "str", type_name: "str"):
    passenger = models.Passenger.objects.filter(name=name).first()
    airplane = models.Airplane.objects.filter(type=type_name).first()
    assert_that(
        airplane.passengers.filter(name=name).first().name
    ).is_equal_to(name)
```

Quelltext 5: Steps-Datei für die Implementierung von Tests mit Behave in Python3

[3] https://github.com/cucumber/cucumber/wiki/Gherkin (10.6.2017)
[4] https://cucumber.io (10.6.2017)

Um Behave Tests übersichtlicher zu gestalten und Funktionsannotationen von Python3 zu nutzen, wurde im Quelltext 5 ein vom Standard abweichender Matcher (*Goat*[5] verwendet. Dieser ermöglicht es, implizite Parameter zu erzeugen und so die von Behave benötigte *Context*[6] Instanz als ersten Funktionsparameter auszulassen. Im Quelltext 5 konnte deshalb das instantiierte Airplane-Objekt von der `given` Methode an die `when` Funktion übergeben werden. Standardmäßig müssten solche Objekte im Kontext von Behave gespeichert und ausgelesen werden. Dies führt bei der Wiederverwendung von Sätzen häufig zu Attributfehlern. Durch Goat werden für die Funktion benötigte Instanzen klar durch die Funktionsparameter definiert und sind so unabhängig vom Behave Kontext.
Zusätzlich zur Aufgabenstellung mussten für den Editor+ Konfigurationsdateien geschrieben werden. Da die Filterung des Quellcodes mit Hilfe der Tests im ersten Prototypen nicht enthalten ist, wurden die Tests manuell vom Evaluationsleiter analysiert. Für die erste Emulation des Features wurden alle Klassen und Funktionen ausgeblendet, die nicht in den Szenario Steps der Behave Tests vorkommen. Steps, die im Background[7] Abschnitt der Feature-Datei aufgerufen werden, sind nicht betrachtet wurden. Zusätzlich wurde der Task-Distributor emuliert und einzelne Funktionen durch die Konfigurationsdatei mit kontextspezifischen Informationen angereichert und wenige Elemente wieder eingeblendet. Beide zusätzlichen Aktionen sollen ein *Best-Case* Szenario widerspiegeln, welches zusätzliche Interaktionen des Task-Distributor erfordert. Die zusätzlich ausgeführten Aktionen fließen in die Auswertung ein.

6.3. DURCHFÜHRUNG

Die Evaluation wurde mit sieben Probanden durchgeführt. Alle Teilnehmer sind männlich (Durchschnittsalter: 28,7) und gaben an, mindestens mittlere Programmierkenntnisse in Python und Erfahrungen mit Entwicklungsumgebungen zu besitzen. Drei der Probanden sind professionelle Softwareentwickler im Bereich Python-Entwicklung (Proband 2, 6, 7) mit mehr als sieben Jahren Berufserfahrung. Zwei Teilnehmer haben je fünf Jahre Erfahrung als freiberufliche Softwareentwickler (Proband 1 und 4). Ein weiterer Teilnehmer (Proband 3) gab an, weniger als ein Jahr Berufserfahrung (Bachelorabschluss in Medieninformatik) und nur mittlere Erfahrung mit Python zu besitzen. Proband 5 ist Quereinsteiger (abgeschlossenes Biologie-Studium), hat keine Berufserfahrung und wenig Programmiererfahrung.
Die Evaluation fand an unterschiedlichen Orten statt. Es wurde darauf geachtet, dass Probanden die Evaluation ohne störende äußerliche Einflüsse durchführen konnten. Die Bearbeitung der Implementierungsaufgaben und die Beantwortung des Fragebogens erfolgten dabei mit einem bereitgestellten Laptop. Die Uhrzeit wurde durch die Teilnehmer bestimmt. Bei der Durchführung wurden die Probanden von zwei Experten beobachtet, welche bei Problemen befragt werden konnten.

[5] `https://github.com/cuescience/goat` (20.8.2017)
[6] *Context* ist eine Klasse von Behave, welche Informationen zum aktuell ausgeführten Testlauf über die einzelnen Testfunktionen hinweg speichert. Diese wird häufig genutzt um im Testlauf erstellte Instanzen global erreichbar zu machen `https://pythonhosted.org/behave/api.html#behave.runner.Context` (13.6.2017)
[7] `https://pythonhosted.org/behave/gherkin.html#background` (12.8.2017)

6.4. AUSWERTUNG

Der folgende Absatz fasst die Ergebnisse der Evaluation zusammen. Hierfür wird das Vorgehen in den einzelnen Editoren für die jeweilige Aufgabe gesondert beschrieben und verglichen. Die Probanden konnte jede Programmieraufgabe ohne große Unterstützung lösen. Durch die geringe Teilnehmeranzahl konnte nicht jede Aufgabe gleich oft in einem Editor gelöst werden. Für die Berechnung der TLX-Werte wurde deshalb die Gewichtung eingerechnet. Im TLX-Fragebogen spiegeln hohe Zahlenwerte eine starke Belastung (negativ) und niedrige Werte eine geringe Belastung (positiv) wider. Die **FizzBuzz** Aufgabe war für die Probanden im Durchschnitt die leichteste Aufgabe. Die Programmieraufgaben **Semiplan** und **Booking tickets for flights** wurden ähnlich fordernd bewertet.

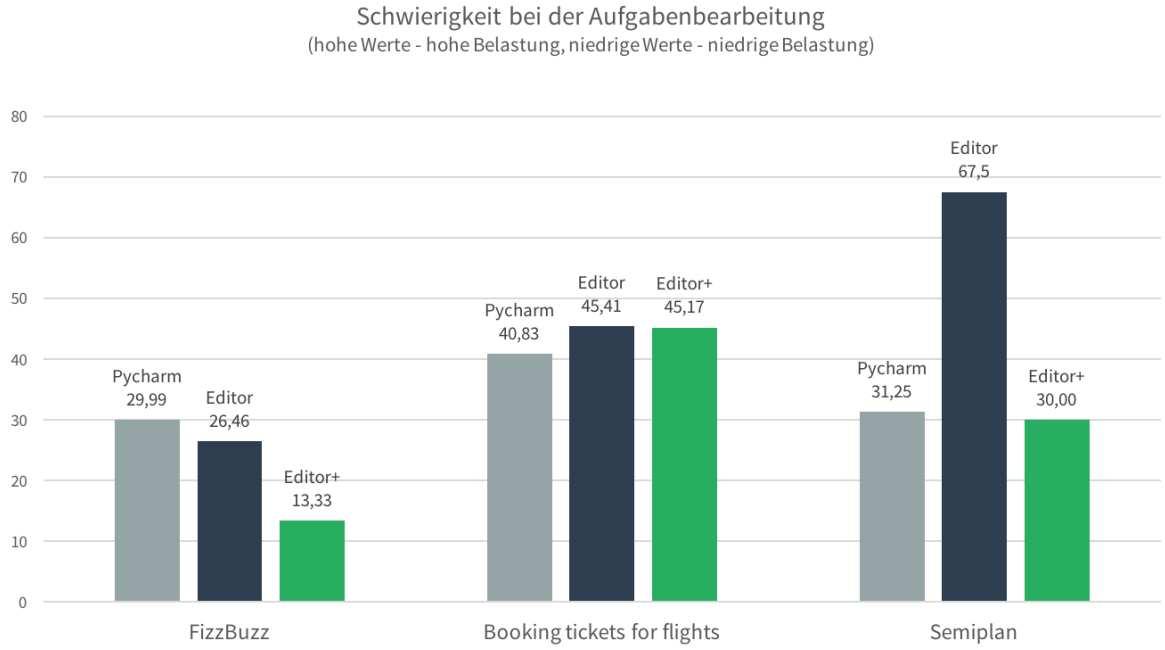


Abbildung 6.2.: Vergleich der Bewertung für die Belastung bei der Lösung einer Aufgabe innerhalb eines Editors

6.4.1. PYCHARM

Die Entwicklungsumgebung Pycharm wurde gewählt, um zu beobachten, wie Probanden in einer etablierten Entwicklungsumgebung Programmieraufgaben lösen. Alle Teilnehmer gaben an, Pycharm als priorisierte Entwicklungsumgebung für Python Programmierung zu verwenden. Die Probanden sollten weiterhin angeregt werden, Vergleiche zwischen Pycharm und dem implementierten Editor dieser Arbeit zu äußern. Hierfür wurde zusätzlich beobachtet, welche Funktionalitäten die Probanden häufig in einer IDE verwenden.
Trotz der guten Erfahrungen mit Entwicklungsumgebungen (Durchschnitt: 4,25 von 5 angegeben) gab es speziell bei der Auswertung von Testausführungen innerhalb von Pycharm Missverständnisse. So wurden erfüllte Tests übersehen oder Fehlerausgaben wegen ihrer Komplexität überflogen.

Keiner der Probanden öffnete die Beschreibung in einem gesonderten Fenster, sodass Aufgabenbeschreibung und Implementierung auf einen Blick erkennbar gewesen wären. Aus diesem Grund kam es zu einem erhöhtem Navigationsaufwand.

FIZZBUZZ

Entgegen den Erwartungen wurde die FizzBuzz Aufgabe in Pycharm fordernder wahrgenommen, als im Editor und Editor+ (Abbildung 6.2). Vor allem Proband 2 hatte einen erhöhten Navigationsaufwand (Körperliche Anforderungen: 55; Anstrengung: 60; Frustration: 55). Erklärbar ist diese Bewertung anhand der wiederholten Ausführung der Tests. Erst nach mehreren Iterationen waren alle Tests erfüllt und der Proband mit seiner Leistung zufrieden (Leistung: 5). Proband 1 und 5 hatten keine Probleme mit der Lösung der Aufgabe in Pycharm. Proband 5, mit der geringsten Programmiererfahrung, nutzte häufig die automatische Vervollständigung der Entwicklungsumgebung und merkte an, dass im Vergleich zu den anderen beiden Editoren, das Erscheinen der Vorschläge ohne Tastenkombination hilfreich ist.

BOOKING TICKETS FOR FLIGHTS

Die Booking tickets for flights Aufgabe erhielt in Pycharm eine bessere durchschnittliche Bewertung als im Editor+. Die beiden aufgenommenen Ergebnisse gehen dabei stark auseinander. Proband 3 bewertet die Aufgabe mit 60 deutlich höher als Proband 6 (Durchschnittswert: 21,66). Diese Ergebnisse spiegeln sich in den unterschiedlichen Programmiererfahrungen der beiden Entwickler wider. Proband 3 ist deutlich unerfahrener in der Entwicklung von Django-Anwendungen als Proband 6. Beide Probanden implementierten schon vorhandene Funktionalitäten, da sie diese im Quelltext übersehen hatten.

SEMIPLAN

Die Semiplan Aufgabe wurde in Pycharm ebenfalls von zwei Probanden gelöst. Proband 4 tat sich dabei deutlich schwerer als Proband 7. Beide hatten einen erhöhten Navigationsaufwand innerhalb der Dateien, um relevante Quelltextabschnitte zu finden. Proband 7 wollte, anstatt der vorhandene Methode `is_cancelable` eine eigene Methode hinzufügen. Dies wurde durch den Leiter der Evaluation unterbunden. Bei Proband 4 spiegelte sich der erhöhte Suchaufwand im Wert der körperlichen Anforderung wider (80). Proband 7 bewertete die körperliche Anforderung in allen drei Editoren mit dem gleichen Wert (5). Diese Bewertung könnte durch eine fehlende Erläuterung des Kriteriums entstanden sein. Eigentlich wurde den Probanden dieser Wert als *Klick- und Navigationsaufwand* vorgestellt. Weiterhin benötigte Proband 7 nur eine Testausführung um die Aufgabe zu erfüllen, was zu einer sehr positiven Gesamteinschätzung des Probanden führte (Leistung: 5; Frustration: 5).

6.4.2. EDITOR

Der Editor ohne Hilfen erhielt im Durchschnitt schlechtere Bewertungen als Pycharm. Dass der Editor bei der FizzBuzz Aufgabe besser bewertet wurde, ist wahrscheinlich dem direkten Einstieg in die Aufgabe geschuldet. So konnten die Probanden im Editor der Plattform sofort mit der Implementierung beginnen, wohingegen in Pycharm zunächst die Readme und Dateien geöffnet werden mussten. Weiterhin lobten Probanden, dass alle benötigten Informationen auf einen Blick sichtbar waren. Das fehlende Hervorheben der zu bearbeitenden

Funktion führte zu einem erhöhten Suchaufwand. Entgegen der Erwartungen wurden keine Änderungen an unzulässigen Stellen des Quellcodes durchgeführt. Proband 5 traute dem Editor keine Autovervollständigung zu. Dieser Umstand war der automatischen Darstellung der Codevorschläge, ohne Zutun des Nutzers, in Pycharm geschuldet. Nach einer Erklärung durch den Leiter der Evaluation nutzte Proband 5 die automatische Vervollständigung auch im Editor. Zwei Probanden nutzten die Möglichkeit, Tests von der Anwendung automatisch ausführen zu lassen und bewerteten das Feature als angenehm. Fünf von sieben Probanden erkannten nicht, dass die Test-Szenarien auf der rechten Seite der Anwendung ausklappbar sind (Abbildung 5.7). Dies zeigt ein deutliches Problem in der Gestaltung des User-Interfaces und muss in einer weiteren Iteration ausgebessert werden.

FIZZBUZZ

Die FizzBuzz Aufgabe wurde im Editor von vier Probanden bearbeitet. Dabei zeigen die Ergebnisse ein durchweg positives Ergebnis. Nur Proband 7 bewertete die Bearbeitung der Aufgabe schwieriger. Dem Anschein nach hatte der Proband Probleme mit der Aufgabenstellung. Er verließ zwischenzeitlich den Editor, um im Internet zu recherchieren.

BOOKING TICKETS FOR FLIGHTS

Die Booking tickets for flights Aufgabe wurde zweimal im Editor gelöst. Trotz der fehlenden Hilfen (Ausblenden von Funktionen) hatten die Probanden wenig Probleme bei der Lösung. Beide Teilnehmer übersahen Funktionalitäten, welche im Grundgerüst schon vorhanden waren und implementierten diese selbstständig. Bei wiederholtem Betrachten des Quellcodes fiel beiden die Dopplung auf und sie korrigierten ihre Lösung.

SEMIPLAN

Semiplan wurde nur einmal im Editor bearbeitet. Proband 1 hatte Schwierigkeiten beim Auffinden relevanter Codeabschnitte. So musste der Proband 943 Zeilen Quelltext im Editor durchsuchen (im Editor$^+$ durch das Ausblenden nur 64 Zeilen). Warum die empfundene Belastung viel höher als bei der Bearbeitung in Pycharm ist, kann nicht genau begründet werden. Auslöser könnten die falsche Verwendung der Suche (Suchfunktion des Browsers verwendet anstatt der des Editors), das zusätzliche horizontale Scrollen (durch Länge des Quellcodes) oder das Syntax-Highlighting gewesen sein.

6.4.3. EDITOR$^+$

Der Editor$^+$ erhielt von den Probanden viel positive Resonanz (Abschnitt B). Die Ergebnisse des TLX-Fragebogens spiegeln dieses Feedback nicht voll und ganz wider. Zwar wurde der Editor$^+$ durchweg besser als der Editor ohne Hilfen bewertet, im Vergleich zu Pycharm sind die Bewertungen aber nur geringfügig besser - bei Booking tickets for flights sogar schlechter. Positiver fällt das Ergebnis aus, wenn man den Vergleich innerhalb einer Evaluation betrachtet. Dabei erhielt die Bearbeitung im Editor$^+$ häufig eine bessere Bewertung vom Probanden als die Bearbeitung einer Aufgabe in Pycharm oder dem Editor ohne Hilfen. Gerade beim Vergleich der ähnlich fordernden Aufgaben (Booking tickets for flights und *Semiplan*) ist eine positive Tendenz klar ersichtlich (Abbildung B.2).

Negativ aufgefallen ist, dass die zusätzlichen Kontextinformationen kaum genutzt bzw. gefunden wurden. So merkten Probanden an, dass diese mehr Hervorgehoben werden sollten (beispielsweise durch ein kleines Symbol). In einer weiteren Iteration werden diese zusätzlich in der Navigationskomponente dargestellt. Der Bedarf an kontextbezogenen Informationen war in der Evalutaion trotzdem zu beobachten. So fragten Probanden was eine *Seminar Participation* ist oder was *überbucht* im gegebenen Kontext bedeutet. Meist konnten diese Fragen durch die natürlichsprachlichen Softwarespezifikationen abgedeckt werden.

FIZZBUZZ

Bei der FizzBuzz Aufgabe bot der Editor$^+$ keine Mehrwerte gegenüber dem Editor ohne Hilfen. Dies ist dem nicht vorhanden Codegrundgerüst der Aufgabe geschuldet. Aus diesem Grund wurde diese Konstellation (Editor$^+$ - FizzBuzz) nur zur Absicherung evaluiert. Die deutlich besseren Bewertungen sind durch die mehrmalige Bearbeitung der gleichen Aufgabe erklärbar. Bei der Evaluation traten keine Probleme auf.

BOOKING TICKETS FOR FLIGHTS

Die Booking tickets for flights Aufgabe wurde im Editor$^+$ drei mal gelöst. Im Vergleich zu den anderen beiden Editoren, wurden die benötigte Funktionen schneller gefunden und Funktionalitäten nicht doppelt implementiert. Trotz einer guten Performance und einer schnellen Lösung, bewertete Proband 7 die Aufgabenbearbeitung im Editor$^+$ deutlich fordernder als die zuvor durchgeführte Aufgabe in Pycharm (Semiplan). Damit blieb er der einzige Proband, welcher bei einer komplexen Aufgabe (ausgeschlossen FizzBuzz) stärker im Editor$^+$ gefordert war.

SEMIPLAN

Durch die Funktionalitäten des Editor$^+$ bestand die Aufgabe Semiplan aus 64 Zeilen Code. Im Vergleich zu Pycharm fiel die durchschnittliche Bewertung trotzdem nicht viel positiver aus (1,25 Punkte Unterschied). Für eine aussagekräftige Statistik fehlen hier noch weitere Probanden. Ansonsten hatten Teilnehmer wenig Probleme mit der Lösung der Semiplan Aufgabe innerhalb des Editor$^+$. Die höchste Belastung verspürte Proband 2 (51,66). Dieser bewertete alle anderen Aufgaben aber noch fordernder.

6.5. FAZIT DER EVALUATION

Die Auswertung der Evaluation des Konzepts weist auf ein positives Ergebnis hin. Alle sieben Probanden bewertetet den Editor$^+$ als hilfreich und würden ihn für die Lösung kleiner Aufgaben nutzen. Die technische Umsetzung des Prototypen konnte durch die Beobachtung im praktischen Einsatz und unter realen Bedingungen ebenfalls evaluiert werden. Dabei lief der Prototyp stabil und verursachte nur wenige Probleme, welche die Einschätzung der Probanden beeinflusst haben könnte.
Entgegen den Erwartungen, erleichterte der Editor die Bearbeitung sogar bei Aufgaben ohne Code-Skelett und *Projektdomäne*. So erhielt der Editor bei der FizzBuzz Aufgabe bessere Wertungen als Pycharm. Dies ist hervorzuheben, waren die Probanden doch alle Vertraut im Umgang mit der Pycharm-Entwicklungsumgebung. Ein Grund für das bessere Ergebnis

könnte die kompakte Darstellung aller benötigten Informationen (Aufgabenstellung, Tests, Editor) sein. So hatten die Probanden einen geringeren Navigationsaufwand, welcher sich in der durchschnittlichen Bewertung der körperlichen Anforderungen widerspiegelt (Pycharm: 26,42 und Editor$^+$: 18,89). Weiterhin positiv benannten die Probanden das Ausblenden von Klassen und Funktionen. Bei der eigentlichen Umsetzung fiel dieses Feature den Probanden zunächst nicht auf. Um es in die Evaluation einfließen zu lassen, wurde den Probanden zum Schluss der Aufgabenbearbeitung der Quellcode ohne Ausblendung vorgelegt. Die Minimierung der kognitiven Belastung des Crowdworkers im Editor$^+$ hängt dabei stark von der Filterung des Code-Skeletts durch die Tests ab. In der Evaluation wurde diese Filterung manuell durchgeführt. Dabei wurden Klassen und Methoden, welche nicht in den Tests vorkamen, ausgeblendet. Weiterhin wurden wenige Funktionen vom Aufgabenersteller wieder eingeblendet. Dies war notwendig, um Probanden ohne Framework-spezifisches Wissen (Django), die Bearbeitung zu ermöglichen. In einer weiteren Iteration muss das Erstellen der Konfigurationsdatei automatisiert erfolgen. Hierfür muss die Analyse der vorliegenden Tests implementiert werden. Diese wird starken Einfluss auf die Nützlichkeit des erarbeiteten Konzepts haben. Probleme könnten dabei die Testimplementierung des Task-Distributor bereiten. Benutzt dieser viele Klassen und Module innerhalb der Test-Implementierungen, werden dem Crowdworker eventuell viele unnötige Abschnitte präsentiert.
Negativ fiel die Irrelevanz der zusätzlichen Kontextinformationen auf. Entweder wurden diese vom Probanden nicht gefunden oder sie wurden überlesen. Die Notwendigkeit äußerte sich durch Fragen der Probanden. Aus diesem Grund müssen die zusätzlichen Informationen eine andere Visualisierung erhalten. Eine Möglichkeit wäre das Hervorheben von angereicherten Klassen und Funktionen innerhalb des Editors. Abschließend werden in Tabelle 6.1 die Evaluationsergebnisse mit den Anforderungen dieser Arbeit (Tabelle 2.1) abgeglichen.

MUST		
Analyse		
M10	Aufgabentitel und Aufgabenbeschreibung müssen verständlich dargestellt werden	**erfüllt** Aufgabentitel und Aufgabenbeschreibung werden im Webinterface visualisiert. Die Verständlichkeit ist weiterhin abhängig vom Task-Distributor. Durch die strikten Vorgaben des bewährten *Behavior Driven Development* wurden alle Aufgaben in der Evaluation verstanden und gelöst.
Implementierung		
M20	Verfassen von Quellcode muss möglich sein	**erfüllt** Aufgaben konnten von den Probanden angenommen und bearbeitet werden. Das umgesetzte Webinterface bot die Möglichkeit, Quellcode zu implementieren. Für die Umsetzung des Editors wurde das *Eclipse Orion Code Edit Widget* verwendet (Kapitel 3.2.1). Die Persistenz wird von *Eclipse Che* sichergestellt (Kapitel 3.2.2).

M30	Nur für den Nutzer freigegebene Dateien dürfen zugänglich sein	**erfüllt** Durch den implementierten File-Proxy konnten in der Evaluation Dateien, Klassen und Funktionen ausgeblendet werden. Die Anforderung wird somit als erfüllt angesehen. Im Prototypen ist diese Funktionalität noch sprachabhängig. Um dieses Kriterium sprachenunabhängig zu erfüllen muss das textbasierte Ausblenden auf AST-Ebene umgestellt werden.
Feedback		
M40	Implementierter Quellcode muss ausgeführt bzw. getestet werden können	**erfüllt** Die Ausführung des Quellcodes und der Tests wird durch Eclipse Che gesichert. Diese Funktionalität ist durch die Nutzung der Virtualisierungstechnologie Docker sprachenunabhängig. Somit können jegliche Laufzeitumgebungen abgebildet werden.
M50	Ausgabe der Testausführung muss textuell dargestellt werden	**erfüllt** Die Ergebnisse der Testausführung wurden zunächst im Prototypen textuell ausgegeben und dargestellt. Hierfür verbindet sich der Prototyp mit der Command-API von Eclipse Che und erhält das Ergebnis über eine Websocket-Verbindung.
SHOULD		
Analyse		
S10	Programmverstehen für den Crowdworker erleichtern (vgl. E1 - E7 Abbildung 2.1)	**teilweise erfüllt** Bei der Umsetzung des Prototypen wurde zunächst auf die Implementierung des Editors und die Unterstützung des Crowdworkers beim Bottom-Up Programmverstehens Wert gelegt. Die erste Evaluation zeigt eine positive Tendenz im Vergleich zu häufig verwendeten Entwicklungsumgebungen (durchschnittliche Geistige Anforderung in Pycharm: 46,25; im Editor$^+$: 41,42). Um ein aussagekräftigeres Ergebnis zu erhalten, sollte die Evaluation mit weiteren Probanden durchgeführt werden. Weiterhin muss die Navigationskomponente des Konzepts umgesetzt werden, um das Bottom-Up Vorgehen der Entwickler zu unterstützen.
Implementierung		
S20	Nutzer bei der Implementierung unterstützen	**erfüllt** Damit der Nutzer bei Implementierung unterstützt wird, unterstützt der Editor des Prototypen das Language-Server-Protokoll. Somit erhält der Nutzer die Möglichkeit, Hover-Tooltips und Autovervollständigung zu nutzen. Über eine Konfigurationsdatei kann der Task-Distributor zusätzliche Kontextinformationen hinzufügen. Um diese Informationen dem Crowd-Entwickler einfacher zugänglich zu machen, sollte diese in der nächsten Umsetzung mehr hervorgehoben werden.

Feedback S30	Fehlerausgaben müssen verständlich dargestellt werden	erfüllt Die Verständlichkeit der Ausgaben der Test*Assertions* sind abhängig vom Task-Distributor. Im Gegensatz zu Pycharm war durch die klare Zuordnung der Fehlerausgabe zum entsprechenden Test-Satz (Step) das Feedback schneller zuzuordnen.
COULD		
Analyse C10	Verschiedene Abstraktionsebenen des Quellcodes visualisieren	noch nicht erfüllt Verschiedene Abstraktionsebenen des Quellcodes zu visualisieren, um dem Crowd-Entwickler beim Top-Down Programmverstehen zu unterstützen, muss in der nächsten Iteration fokussiert werden. Denkbar wären alle Formen von Diagrammen, welche der Task-Distributor in der Analysephase für das anstehende Softwareprojekt entwirft. Automatisiert darstellen ließen sich UML-Diagramme. Diese könnten mit den Daten aus der Konfigurationsdatei angereichert werden.
Implementierung C20	Refactoring, automatische Formatierung und GoTo-Funktionalität für Python, PHP und Javascript	teilweise erfüllt Zusätzliche Funktionalitäten und die Sprachenunabhängigkeit sind durch die Verwendung des Language-Server-Protokolls gesichert. Dabei wurde zunächst die automatische Formatierung implementiert. Da der Nutzer auf einen Bereich in der Datei eingeschränkt ist, muss das *RangeFormatting* des Langugage-Server-Protokolls implementiert werden. Für Refactoring Funktionalitäten sind aufwändigere Implementationen notwendig. Um diese Anforderung zu erfüllen, müssen in der nächsten Evaluation weitere Programmieraufgaben mit anderen Programmiersprachen evaluiert werden.
C30	Sinnvolle Verlinkung zwischen Code-Implementierung und Code-Exploration (vgl. E11 Abbildung 2.1)	noch nicht erfüllt Da die Navigationskomponente (CodeExploration) noch nicht vollständig umgesetzt wurde, ist diese Anforderung nicht erfüllt. Im Konzept (Kapitel 4.2.1) ist die Verknüpfung beschrieben und wird weiterhin als notwendig betrachtet.
Feedback C40	Kommunikationskomponente um Fehler zu melden oder Fragen zu stellen	noch nicht erfüllt Die Kommunikationskomponente wurde im ersten Prototypen nicht implementiert, da in der Evaluation zunächst nur Crowd-Entwickler als Probanden vorgesehen waren. Die Kommunikation bleibt weiterhin ein kritischer Punkt des Konzepts.

Tabelle 6.1.: Abgleich zwischen den Evaluationsergebnissen und den aufgestellten Anforderungen

7. ZUSAMMENFASSUNG UND AUSBLICK

Das folgende Kapitel fasst die Ergebnisse dieser Arbeit zusammen und bewertet das erarbeitete Konzept sowie den implementierten Prototypen. Weiterhin werden das Vorgehen dieser Arbeit kritisch betrachtet und alternative Herangehensweisen genannt.

7.1. VORGEHEN

Die Arbeit mit der Literaturrecherche zu *Programmverstehen* (Kapitel 2.1) zu beginnen, hatte positiven Einfluss auf die Erarbeitung der Anforderungen. Wie sich bei der *State-of-the-Art-Analyse* verwandter Konzepte zeigte, ist der Kontext eines Softwareprojektes selbst bei kleinen Programmieraufgaben nicht immer vernachlässigbar. Den Entwickler beim Programmverstehen zu unterstützen sollte ein essentieller Bestandteil einer Crowd-basierten Plattform für Softwareentwicklung sein.
Durch die Recherche nach bestehenden Kriterien für die Unterstützung beim Programmverstehen konnten erste Anforderungen für das eigene Konzept übernommen werden. Weiterhin wurde durch eine *HTA-Analyse* (Abbildung 2.2) das Vorgehen im Kontext einer *testgetriebenen CrowdsourcingPlattform* untersucht und weitere Anforderungen spezifiziert (Tabelle 2.1).
Um die Anforderungen umzusetzen, wurden existierende *Web-Code-Editoren* und andere Technologien betrachtet (Kapitel 3.1 und 3.2). Weiterhin wurde nach ähnlichen Konzepten recherchiert. Dabei wurde der Fokus auf deren Evaluationen gelegt, um Fehler in der eigenen Umsetzung zu umgehen. So konnten in beiden Konzepten Schwachstellen festgestellt werden. Zum einen führte die schwankende Qualität der Aufgabenspezifikationen zu einem erhöhten Kommunikationsaufwand zwischen Task-Distributor und Crowdworker, zum anderen gingen beide Konzepte davon aus, dass durch *Microtasks* der Kontext des Softwareprojekts für den Crowdentwickler irrelevant wird. Speziell in der Evaluation von *CrowdCode* (Kapitel 3.3.2) merkten Nutzer aber eine erhöhte kognitive Belastung an und forderten eine Übersicht über zu bearbeitende Projekte.
Für die eigene Konzeption wurde zunächst vom Kontext einer *testgetriebenen Crowdsourcing-Plattform* in Form der *polylith-Plattform* (Kapitel 4.1) ausgegangen. Diese wurde in der Konzeption um ein Webinterface erweitert, in dem Crowdworker Aufgaben innerhalb eines Softwareprojektes bearbeiten können. Dabei wurde auf die Erfüllung der aufgestellten An-

forderungen geachtet und erste Entwürfe für das User-Interface erstellt. Für ein besseres Ergebnis hätten zunächst weitere Mock-Ups erarbeitet und früher Probanden vorgestellt werden sollen. So hätten einfache Gestaltungsfehler, wie das Hervorheben ausklappbarer Szenarios, früher aufgedeckt werden können.
Für die Umsetzung der Entwicklungsumgebung wurden *Eclipse Che* und das *Orion Code Edit Widget* verwendet, welche einen Großteil der technischen Anforderungen erfüllten. Das *Language-Server-Protokoll* ermöglichte eine schnelle Implementierung von Unterstützungen für den Crowd-Entwickler.
Für die Evaluation wurden eine etablierte und zwei eigene Programmieraufgaben verwendet. Durch den fortgeschrittenen Stand des Prototypen konnten die erarbeiteten Aufgaben einfach in die Anwendung eingefügt und evaluiert werden. Bei der geführte Evaluation konnten Wünsche von Probanden beobachtet und aufgenommen werden. Für eine fundiertere und aussagekräftigere Evaluation hätten weitere Probanden evaluiert werden sollen. Vor allem die ungleichmäßige Verteilung der Aufgaben auf die einzelnen Editoren führte zu einer gewichteten Berechnung der Ergebnisse.

7.2. ERGEBNISSE DER ARBEIT

Die Evaluation des Prototypen spiegelt eine positive Tendenz für das erarbeitete Konzept wider. Neben den technischen konnten auch nichttechnische Anforderungen evaluiert werden. Um die Ergebnisse dieser Arbeit zu verifizieren, sollten diese durch eine weitere Evaluation mit einer größeren Anzahl an Probanden bestätigt werden.
Die Abbildung des *Behaviour Driven Development* auf das erarbeitete Konzept und Webinterface stellte sich somit als eine positive Entscheidung dar. So konnten auch unerfahrenere Entwickler sämtliche Aufgaben im Prototypen lösen. Die Qualität der Aufgabenbeschreibung bleibt dabei weiterhin abhängig vom Task-Distributor. Durch die strikte Einhaltung der in der Praxis etablierten *Given-Then-When* Stuktur wird den Aufgabenerstellern aber eine erfolgversprechende Aufbereitungsmethode angeboten. Die natürlichsprachlichen Tests geben dem Crowd-Entwickler ein stetiges Feedback zum Fortschritt der Aufgabenlösung. Durch das Language-Server-Protokoll konnten dem Crowdworker hilfreiche Funktionalitäten für die Implementierung der Aufgabe bereitgestellt werden. Diese Features sind dabei sprachenunabhängig und können so für fast jede Programmiersprache angeboten werden. Auch die Nutzung von Eclipse Che als Workspace-Server hat sich bewährt und ermöglichte eine schnellere Umsetzung des Prototypen. Weiterhin besitzt der Task-Distributor die Möglichkeit kontextspezifische Informationen hinzuzufügen, um dem Crowdworker beim Programmverständnis Hilfe zu leisten. Die Evaluation zeigte die Notwendigkeit solcher Informationen. In einer weiteren Iteration des Editors sollten diese aber stärker hervorgehoben werden.
Die kognitive Belastung und das Bottom-Up Programmverstehen des Crowdworker konnten durch das Ausblenden von irrelevanten Codeabschnitten verbessert werden. Die Implementierung der Analyse des Codegrundgerüsts mit Hilfe der vorliegenden Tests steht noch aus und wird die Nützlichkeit des Konzepts stark beeinflussen. Auch die im Konzept beschriebene Navigationskomponente muss noch umgesetzt und evaluiert werden. Für die Umsetzung wurde im Rahmen dieser Arbeit das *JointJs-Framework* recherchiert, mit dem verschiedene Diagrammtypen (unter anderem auch UML) in HTML erzeugt werden können.

7.3. AUSBLICK

Das Konzept und der erarbeitete Prototyp dieser Arbeit bilden eine gute Grundlage für weitere Arbeiten. Zunächst sollte das Feedback der Nutzerevaluation eingearbeitet werden, um die *Usability* (Gebrauchstauglichkeit) des Prototypen weiter zu verbessern. Abbildung 7.1 zeigt eine erste Umsetzung der geplanten Verbesserungen. Da die Nützlichkeit des Editors stark von der Filterung des Quellcodes abhängt, sollte eine erste automatisierte Erstellung der Konfigurationsdatei erarbeitet werden. Hierfür sollten geeignete statische Quellcodeanalysen recherchiert und angewandt werden. Um die Unterstützung für weitere Sprachen zu ermöglichen, muss das textbasierte Ausblenden der Dateien auf einen AST-basierten Ansatz umgestellt werden. Dafür kann beispielsweise das Framework *GumTree* in Kombination mit *Antlr*[1] verwendet werden. Letzteres bietet die Möglichkeit für verschiedenste Programmiersprachen den AST generieren zu lassen und diesen von GumTree in eine kompaktere Struktur zu bringen. Für das Ausblenden würden zunächst Start- und Endzeile der Modelle und Funktionen, sowie der Name und Typ des Knoten benötigt werden.
Um komplexere Aufgaben in der Entwicklungsumgebung abdecken zu können, kann über eine Erweiterung des Konzepts nachgedacht werden, sodass der Entwickler die Möglichkeit erhält, an mehreren Funktionen Änderungen vornehmen zu dürfen. Weiterhin muss der Crowd-Entwickler eine Möglichkeit erhalten, zusätzliche Module einzubinden. Dabei muss evaluiert werden, ob dafür eine Anfrage beim Task-Distributor notwendig sein sollte oder der Crowd-Entwickler dies ohne Einschränkung durchführen kann.

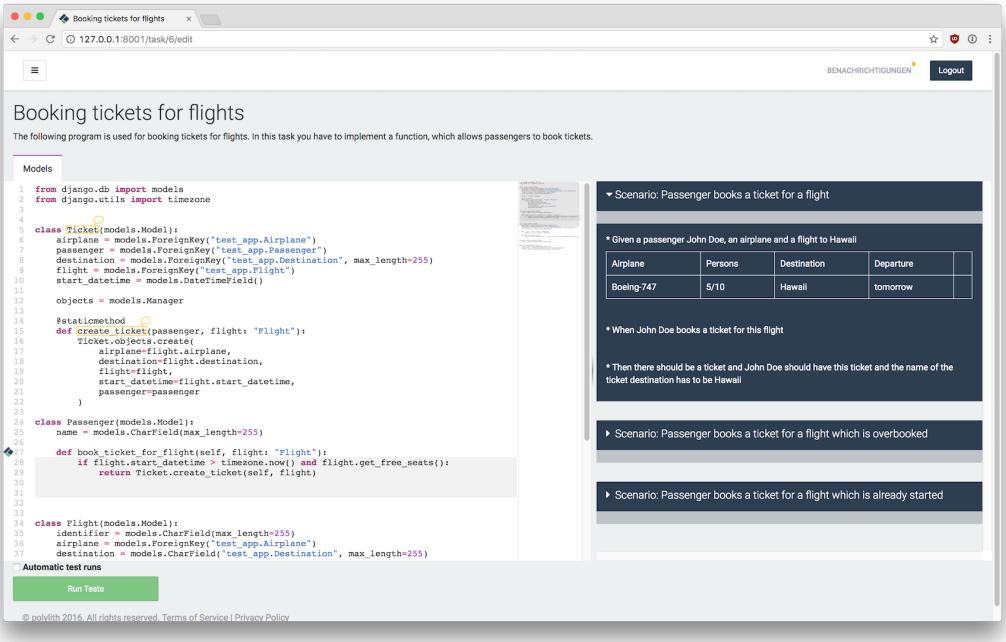


Abbildung 7.1.: Mockup vom Editor mit ersten Verbesserungen aus der Evaluation (angereicherte Informationen hervorgehoben und Test-Szenarios ausklappbar dargestellt)

[1] http://www.antlr.org (21.8.2017)

Ebenso sollte die *Navigationskomponente* des Konzepts vollständig umgesetzt und mit dem Editor verbunden werden. Diese kann das Top-Down Programmverstehen des Crowd-Entwicklers unterstützen. Dabei können die vom Task-Distributor zusätzlich eingepflegten Informationen (Konfigurationsdatei), welche in der ersten Evaluation zu wenig Beachtung fanden, stärker hervorgehoben werden.
Weiterhin ist ein Zusammenschluss mit den Ergebnissen aus den Arbeiten von Hanspach (Gamification-Komponente [Han17]) und Blume (Code-Metriken [Blu17]) geplant, um das Gesamtkonzept der polylith-Plattform evaluieren zu können.

LITERATURVERZEICHNIS

[And06] ANDREA OBILTSCHNIG: Einführung in das wissensbasierte Verstehen von Programmen. In: *Konferenz-Seminar aus Software-Engineering Universität Klagenfurt* (2006), S. 28–38. ISBN 1586036408

[Bäc] BÄCHLE, Michael.: *Wissensmanagement mit Social Media : Grundlagen und Anwendungen*. https://www.degruyter.com/viewbooktoc/product/430631

[Bau13] BAUER, Ilja: Entwurf und Realisierung einer Schnittstelle für die Bindung von Natürlichsprachlichen Softwarespezifikationen an Programmiersprachen. (2013)

[BE94] BALL, Thomas ; EICK, Stephen G.: Visualizing program slices. In: *Visual Languages, 1994. Proceedings., IEEE ...* (1994), Nr. October, 288–295. http://dx.doi.org/10.1109/VL.1994.363606. – DOI 10.1109/VL.1994.363606. – ISBN 0818666609

[Bec03] BECK, Kent: Test-Driven Development By Example. In: *Rivers* 2 (2003), Nr. c, 176. http://dx.doi.org/10.5381/jot.2003.2.2.r1. – DOI 10.5381/jot.2003.2.2.r1. – ISBN 0321146530

[Blu17] BLUME, Jakob: *Bewertung der Kodequalität in einer testgetriebenen Crowdsourcing-Plattform*. 2017. – unveröffentlicht

[Bro83] BROOKS, Ruven: Towards a theory of the comprehension of computer programs. In: *International Journal of Man-Machine Studies* 18 (1983), Nr. 6, 543–554. http://dx.doi.org/10.1016/S0020-7373(83)80031-5. – DOI 10.1016/S0020–7373(83)80031–5. – ISBN 0020–7373

[Bus98] BUSCHMANN, Frank: *Pattern-orientierte Software-Architektur: Ein Pattern-System*. Pearson Deutschland GmbH, 1998

[EPS14] EITELJÖRGE, Florian ; PROF, Prüfer ; SCHNEIDER, Kurt: *Konzept und Implementierung einer aufgabenfokussierten Testumgebung für eine Crowdsourcing Plattform*, Diss., 2014

[FMB+14] FALLERI, Jean-Rémy ; MORANDAT, Floréal ; BLANC, Xavier ; MARTINEZ, Matias ; MONPERRUS, Martin: Fine-grained and accurate source code differencing. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, 313–324

[Gol12] GOLDMAN, Max: *Software Development with Real-Time Collaborative Editing*, Massachusetts Institute of Technology, Diss., 2012

[Gro04] GROSS, Barbara-Ulrike: *Bestimmung von Schwierigkeitsgraden in einer zu entwickelnden Versuchsumgebung Diplomarbeit*, Humboldt-Universität zu Berlin, Diss., 2004

[Han17] HANSPACH, Felix: *Entwicklung eines Gamification-Konzepts für Crowd-basierte Softwareentwicklung*. 2017. – unveröffentlicht

[HS88] HART, Sandra G. ; STAVELAND, Lowell E.: Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In: *Advances in psychology* 52 (1988), S. 139–183

[JSO10] JSON-RPC WORKING GROUP: *JSON-RPC 2.0 Specification*. `http://www.jsonrpc.org/`. Version: 2010

[KKVV12] KATS, Lennart C.L. L L. ; KALLEBERG, Karl T. ; VOGELIJ, Richard G. ; VISSER, Eelco: Software development environments on the web. In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software - Onward! '12* 3 (2012), 99. `http://dx.doi.org/10.1145/2384592.2384603`. – DOI 10.1145/2384592.2384603. ISBN 9781450315623

[Kra16] KRAMER, Tommi D.: *Entwicklung und Umsetzung eines Entscheidungsunterstützungssystems für das Outsourcing in der komponentenbasierten Softwareentwicklung*, Universität Mannheim, Diss., 2016

[KSK11] KITTUR, Aniket ; SMUS, Boris ; KRAUT, Robert: CrowdForge Crowdsourcing Complex Work. In: *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems - CHI EA '11* (2011), 1801. `http://dx.doi.org/10.1145/1979742.1979902`. – DOI 10.1145/1979742.1979902. – ISBN 9781450302685

[KT09] KOSCHKE, Rainer ; TIARKS, Rebecca: Haben wir Programmverstehen schon ganz verstanden? In: *CEUR Workshop Proceedings* 537 (2009), S. 15–26. – ISSN 16130073

[LA94] LEUNG, Y. K. ; APPERLEY, M. D.: A review and taxonomy of distortion-oriented presentation techniques. In: *ACM Transactions on Computer-Human Interaction* 1 (1994), Nr. 2, S. 126–160. `http://dx.doi.org/10.1145/180171.180173`. – DOI 10.1145/180171.180173. – ISBN 1073–0516

[LBAV14] LATOZA, Thomas D. ; BEN TOWNE, W ; ADRIANO, Christian M. ; VAN DER HOEK, André: Microtask programming: building software with a crowd. In: *UIST '14: Proceedings of the 27th annual ACM symposium on User interface software and technology* (2014), 43–54. `http://dx.doi.org/10.1145/2642918.2647349`. – DOI 10.1145/2642918.2647349. – ISBN 9781450330695

[LS13] LEIMEISTER, Jan ; SHKODRAN, Zogaj: Neue Arbeitsorganisation durch Crowdsourcing. In: *Arbeitspapier 287* (2013)

[Mac14] MACDONALD, Marc: *Orion Documentation Architecture*. `https://wiki.eclipse.org/Orion/Documentation/Developer_Guide/Architecture`. Version: 2014

[MFC01] MACKINNON, Tim ; FREEMAN, Steve ; CRAIG, Philip: Endo-Testing : Unit Testing with Mock Objects. In: *Extreme programming examined* (2001), 287–301. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.3214&rep=rep1&type=pdf. ISBN 0201710404

[Per10] PERCIVAL, Harry: *Test-driven Web Development with Python*. O'Reilly, 2010. – ISBN 9781449364823

[Pre10] PREIM, BERNHARD AND DACHSELT, Raimund: *Interaktive Systeme: Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung*. Springer-Verlag, 2010. – ISBN 9783642054013

[Rob13] ROBERT, Martin: *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. 2013. – ISBN 978–3–8266–5548–7

[SFM99] STOREY, M.-A.D ; FRACCHIA, F.D ; MÜLLER, H.A: Cognitive design elements to support the construction of a mental model during software exploration. In: *Journal of Systems and Software* 44 (1999), Nr. 3, 171–185. http://dx.doi.org/10.1016/S0164-1212(98)10055-9. – DOI 10.1016/S0164–1212(98)10055–9. – ISBN 0–8186–7993–X

[SM79] SHNEIDERMAN, Ben ; MAYER, Richard: Syntactic / Semantic Interactions in Programmer Behavior : A Model and Experimental Results. In: *International Journal of Computer and Information Sciences* 8 (1979), Nr. 3, S. 219–238

[SWD12] SOEKEN, Mathias ; WILLE, R.a ; DRECHSLER, R.a B.: Assisted behavior driven development using natural language processing. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7304 LNCS (2012), 269–287. http://www.scopus.com/inward/record.url?eid=2-s2.0-84862188111&partnerID=40&md5=3f9e6990f0071f032a96c05dea733985

ABBILDUNGSVERZEICHNIS

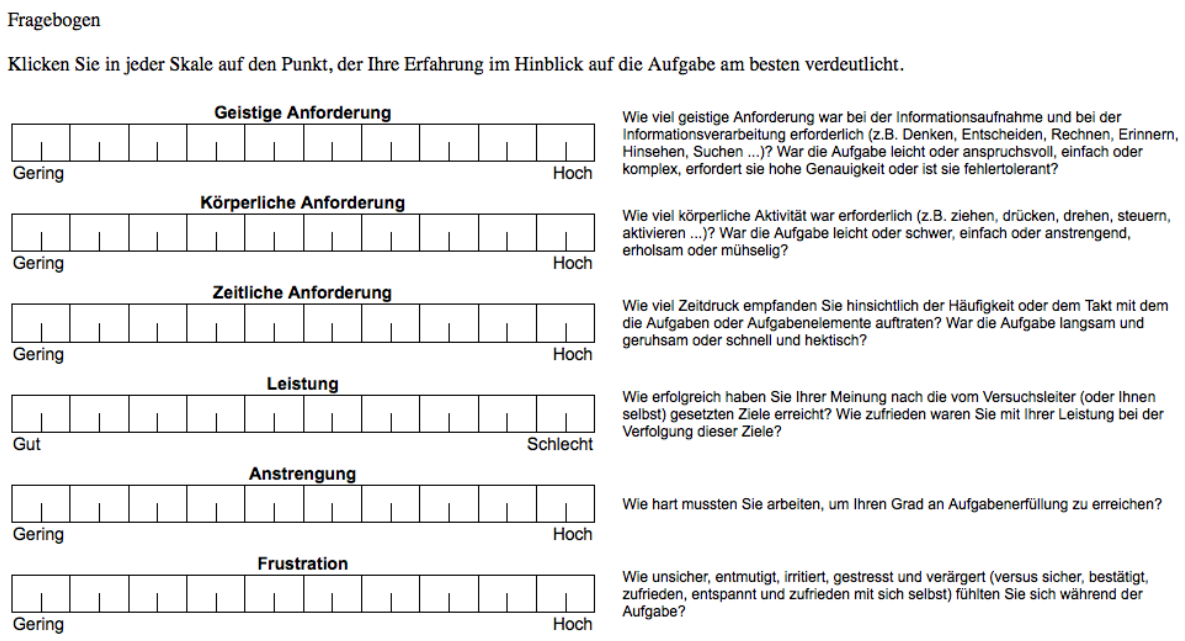
TABELLENVERZEICHNIS

QUELLCODEVERZEICHNIS

A. FRAGEBÖGEN

Im Folgenden werden die verwendeten Fragebögen dargestellt.

A.1. VERKÜRZTER NASA TLX



Fragebogen

Klicken Sie in jeder Skale auf den Punkt, der Ihre Erfahrung im Hinblick auf die Aufgabe am besten verdeutlicht.

Geistige Anforderung

Gering — Hoch

Wie viel geistige Anforderung war bei der Informationsaufnahme und bei der Informationsverarbeitung erforderlich (z.B. Denken, Entscheiden, Rechnen, Erinnern, Hinsehen, Suchen ...)? War die Aufgabe leicht oder anspruchsvoll, einfach oder komplex, erfordert sie hohe Genauigkeit oder ist sie fehlertolerant?

Körperliche Anforderung

Gering — Hoch

Wie viel körperliche Aktivität war erforderlich (z.B. ziehen, drücken, drehen, steuern, aktivieren ...)? War die Aufgabe leicht oder schwer, einfach oder anstrengend, erholsam oder mühselig?

Zeitliche Anforderung

Gering — Hoch

Wie viel Zeitdruck empfanden Sie hinsichtlich der Häufigkeit oder dem Takt mit dem die Aufgaben oder Aufgabenelemente auftraten? War die Aufgabe langsam und geruhsam oder schnell und hektisch?

Leistung

Gut — Schlecht

Wie erfolgreich haben Sie Ihrer Meinung nach die vom Versuchsleiter (oder Ihnen selbst) gesetzten Ziele erreicht? Wie zufrieden waren Sie mit Ihrer Leistung bei der Verfolgung dieser Ziele?

Anstrengung

Gering — Hoch

Wie hart mussten Sie arbeiten, um Ihren Grad an Aufgabenerfüllung zu erreichen?

Frustration

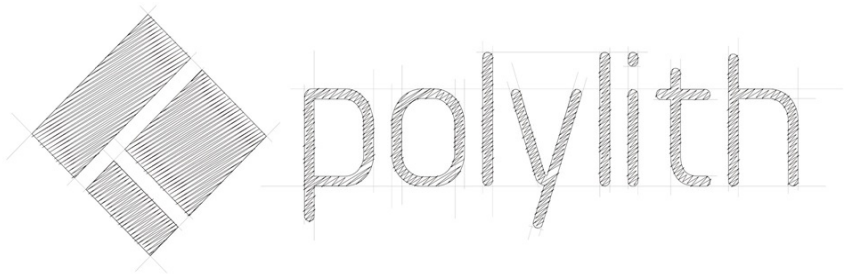
Gering — Hoch

Wie unsicher, entmutigt, irritiert, gestresst und verärgert (versus sicher, bestätigt, zufrieden, entspannt und zufrieden mit sich selbst) fühlten Sie sich während der Aufgabe?

Abbildung A.1.: Online Version des TLX Fragebogens[1]

[1] `http://jensgrubert.bplaced.net/nasa-tlx-short/TLX-German-short.html` (13.8.2017)

A.2. ONLINE FRAGEBOGEN



polylith IDE & Gamification Evaluation

Seite 1

Bewertung des Editors mit Hilfen *

	stimme ich gar nicht zu				stimme ich voll und ganz zu
Ich verstehe wie der Editor funktioniert	◯	◯	◯	◯	◯
Ich fand es hilfreich, dass unrelevante Klassen ausgeblendet werden.	◯	◯	◯	◯	◯
Der Editor hat mir die Arbeit erleichtert.	◯	◯	◯	◯	◯
Ich würde den Editor nutzen um kleine Aufgaben zu entwickeln.	◯	◯	◯	◯	◯

Was ist Dein Verbesserungsvorschlag?

Abbildung A.2.: Erste Seite des Online-Fragebogens: Einschätzung des Editor+

Bewertung der Plattform ohne Gamification

Versuche die Plattform, ohne die Erweiterung durch Gamification-Elemente (Erfolge, Erfahrungspunkte, Ranglisten, usw) aus der Sicht des Crowd-Entwicklers anhand der Kriterien des Octalysis-Systems einzuschätzen. Versuche anzugeben wie stark die einzelnen Kriterien ausgeprägt sind.

Die Plattform besteht aus einer Liste von verfügbaren, annehmbaren Aufgaben (je nach Verfügbarkeit mehr oder weniger Aufgaben). Ich kann immer jeweils eine dieser Aufgaben annehmen. Die Aufgaben sind echte Implementierungsaufgaben von Unternehmen. Für jede Aufgabe erhalte ich eine feste und sichere Bezahlung, wenn ich Quellcode schreibe, der alle Kriterien (Tests) der Aufgabe erfüllt und diesen vor dem Ende der Ausschreibungszeit abgebe. Jede Aufgabe besitzt eine Schwierigkeitsstufe (1-3 Sterne), die angibt wie kompliziert, die zu implementierende Aufgabe ist. Jede Aufgabe kann von einer bestimmten Anzahl Crowd-Entwickler angenommen werden. Die Implementierung der Aufgabe erfolgt komplett im Web-Code-Editor innerhalb der Plattform.

Epische Bedeutung und Berufung *

Es vermittelt dem Nutzer, dass er an etwas größerem teilnimmt oder das er berufen ist zu einer Sache, die nur er lösen kann.

wenig ausgeprägt stark ausgeprägt

Entwicklung und Leistung *

Dies ist der innere Antrieb, den jeder Nutzer hat, um voran zu kommen und sich zu entwickeln.

wenig ausgeprägt stark ausgeprägt

Anregung von Kreativität und Feedback *

Wird der Nutzer angeregt seine Kreativität auszuleben, neues auszuprobieren und erhält er dazu sinnvolles Feedback?

wenig ausgeprägt stark ausgeprägt

Eigentum und Besitz *

Der Nutzer möchte gerne Dinge haben und darüber entscheiden können. Das kann sowohl reale Dinge betreffen wie Geld oder Gegenstände, aber auch virtuelle Güter wie Währungen, Punkte, etc.

wenig ausgeprägt stark ausgeprägt

Sozialer Einfluss und Verbundenheit mit anderen Nutzern *

Hierbei geht es darum wie sehr ein Nutzer mit anderen interagiert. Beispielsweise, dass soziale Elemente hergestellt werden, wie. soziale Akzeptanz oder Rückmeldung, aber auch Wettbewerbe.

wenig ausgeprägt stark ausgeprägt

Abbildung A.3.: Zweite Seite des Online-Fragebogens: Bewertung der Plattform ohne Gamification [Han17]

Knappheit und Ungeduld *

Ist der Antrieb etwas haben zu wollen, weil es sehr selten, exklusiv oder gegenwärtig nicht verfügbar ist.

Bsp. einen Gegenstand, einen Status

wenig ausgeprägt stark ausgeprägt

Unberechenbarkeit und Neugier *

Unverhersehbarkeit von Ereignissen, welche den Nutzer betreffen und die Neugierde beim Versuch genau solche Ereignisse auszulösen.

wenig ausgeprägt stark ausgeprägt

Verlust und Vermeidung *

Die Nutzer wollen ungern etwas verlieren, daher werden sie angetrieben dies zu vermeiden und agieren daher.

Bsp. der Verlust eines Gegenstandes, eines Status oder die Vermeidung von anderen negativen Elementen

wenig ausgeprägt stark ausgeprägt

Seite 3

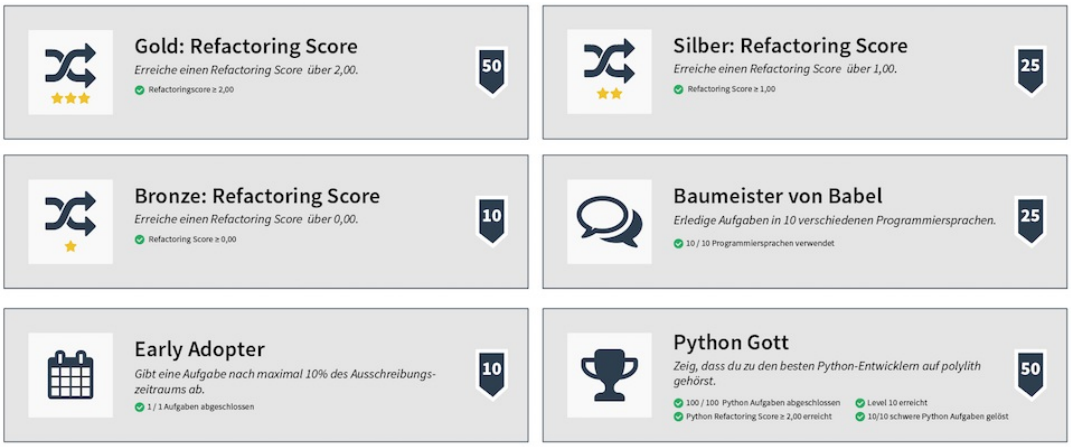
Ein Einsteiger-Erfolg auf der Plattform

Für einige Aktionen auf der Plattform erhält der Nutzer (einmalig) Erfolge. Diese drücken bestimmte Errungenschaften auf der Plattform aus. Einige Erfolge sind sichtbar, bevor man sie errungen hat, für einige müssen die Erfolgskriterien selbst gefunden werden. Für jeden Erfolg erhält der Nutzer Erfolgspunkte (im Beispiel 10). Die Nutzer mit den meisten Erfolgspunkten werden in einer Rangliste dargestellt.

Errungene Erfolge können über Soziale Medien geteilt werden. Weiterhin hat jeder Nutzer ein Profil mit allen errungenen Erfolge.

Abbildung A.4.: Dritte Seite des Online-Fragebogens: Bewertung der Plattform ohne Gamification und Vorstellung eines Erfolges [Han17]

Weitere Beispiele für Erfolge



Was ist deine Meinung zu Erfolgen auf der Plattform? *

	stimme ich gar nicht zu				stimme ich voll und ganz zu
Ich verstehe was dieses Element aussagt.	◯	◯	◯	◯	◯
Ich finde dieses Element optisch ansprechend.	◯	◯	◯	◯	◯
Ich würde versuchen herauszufinden welche Erfolge es zusätzlich gibt.	◯	◯	◯	◯	◯
Ich hätte Interesse daran welche Erfolge andere Benutzer errungen haben.	◯	◯	◯	◯	◯
Ich wäre motiviert weitere Erfolge zu erringen.	◯	◯	◯	◯	◯
Ich hätte Interesse daran, Erfolge zu erringen, die keiner meiner Freunde hat.	◯	◯	◯	◯	◯
Es würde mich demotivieren, wenn ich sehen würde, dass meine Freunde Erfolge haben, die ich nicht habe.	◯	◯	◯	◯	◯

Abbildung A.5.: Vierte Seite des Fragebogens: Evaluation der Erfolge [Han17]

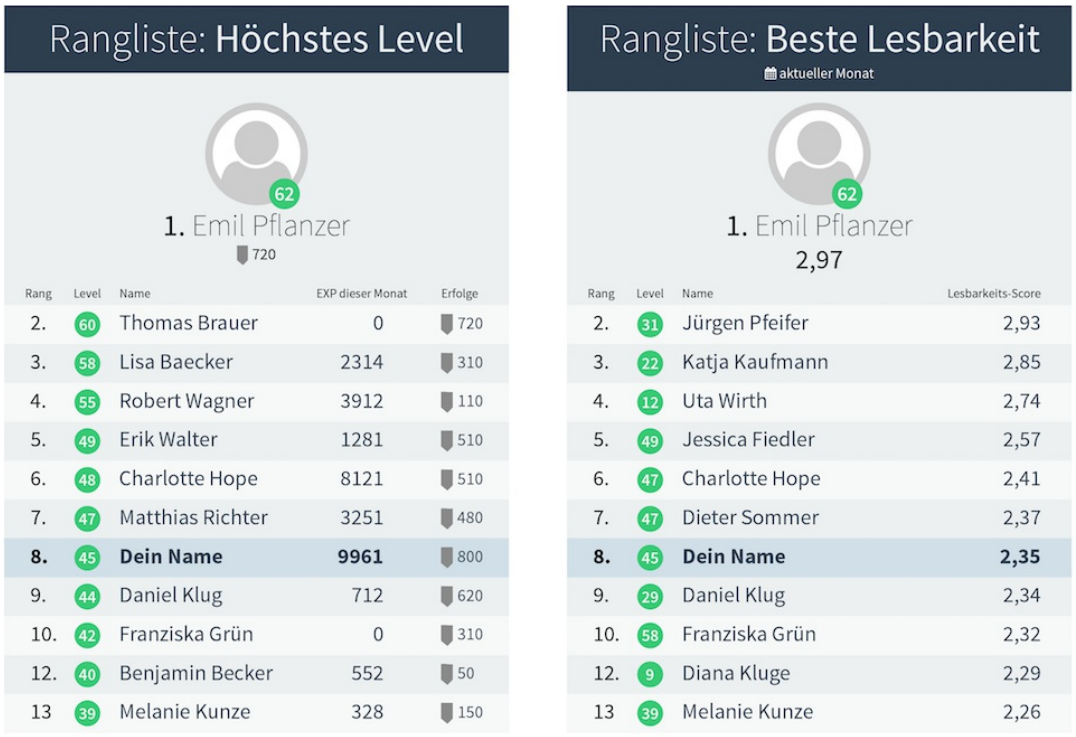
Es würde mich besonders anspornen, wenn ich sehen würde, dass meine Freunde Erfolge haben, die ich nicht habe.	◯	◯	◯	◯	◯
Mich würde es stören, wenn ich für zu viele Aktionen Erfolge bekomme.	◯	◯	◯	◯	◯
Es würde mich stören, wenn meine errungenen Erfolge öffentlich wären.	◯	◯	◯	◯	◯
Es würde mich stören, wenn meine Freunde meine errungenen Erfolge sehen könnten.	◯	◯	◯	◯	◯
Ich finde es spannend, herauszufinden welche Erfolge es noch gibt, indem ich Dinge ausprobiere.	◯	◯	◯	◯	◯

Kommentar

Abbildung A.6.: Fünfte Seite des Fragebogens: Evaluation der Erfolge [Han17]

Seite 4

Entwickler-Ranking anhand des Levels



Rangliste: **Höchstes Level**

62

1. Emil Pflanzer

720

Rang	Level	Name	EXP dieser Monat	Erfolge
2.	60	Thomas Brauer	0	720
3.	58	Lisa Baecker	2314	310
4.	55	Robert Wagner	3912	110
5.	49	Erik Walter	1281	510
6.	48	Charlotte Hope	8121	510
7.	47	Matthias Richter	3251	480
8.	**45**	**Dein Name**	**9961**	**800**
9.	44	Daniel Klug	712	620
10.	42	Franziska Grün	0	310
12.	40	Benjamin Becker	552	50
13	39	Melanie Kunze	328	150

Rangliste: **Beste Lesbarkeit**

aktueller Monat

62

1. Emil Pflanzer

2,97

Rang	Level	Name	Lesbarkeits-Score
2.	31	Jürgen Pfeifer	2,93
3.	22	Katja Kaufmann	2,85
4.	12	Uta Wirth	2,74
5.	49	Jessica Fiedler	2,57
6.	47	Charlotte Hope	2,41
7.	47	Dieter Sommer	2,37
8.	**45**	**Dein Name**	**2,35**
9.	29	Daniel Klug	2,34
10.	58	Franziska Grün	2,32
12.	9	Diana Kluge	2,29
13	39	Melanie Kunze	2,26

Für viele Aktionen auf der Plattform erhält der Nutzer Erfahrungspunkte, mit durch die er ein Level aufsteigen kann. Die Nutzer mit den höchsten Leveln, den meisten Erfolgspunkten und den meisten Erfahrungspunkten innerhalb der letzten Woche/Monat/Jahr können in der Rangliste angezeigt werden.

Abbildung A.7.: Sechste Seite des Fragebogens: Evaluation der Rangliste des Gamification Konzepts [Han17]

Was ist deine Meinung zu Ranglisten auf der Plattform *

	stimme ich gar nicht zu				stimme ich voll und ganz zu
Ich verstehe was dieses Element aussagt.	◯	◯	◯	◯	◯
Ich finde dieses Element optisch ansprechend.	◯	◯	◯	◯	◯
Ich würde versuchen in der Rangliste aufzusteigen.	◯	◯	◯	◯	◯
Es würde mich anspornen einen höheren Rang als meine Freunde zu haben.	◯	◯	◯	◯	◯
Es würde mich demotivieren, wenn ich sehen würde, dass meine Freunde einen höheren Rang haben als ich.	◯	◯	◯	◯	◯
Es würde mich stören, in der Rangliste aufzutauchen.	◯	◯	◯	◯	◯

Abbildung A.8.: Siebte Seite des Fragebogens: Evaluation der Rangliste des Gamification Konzepts [Han17]

Seite 5

Bewertung der abgegebenen Aufgabe anhand der Code-Qualität

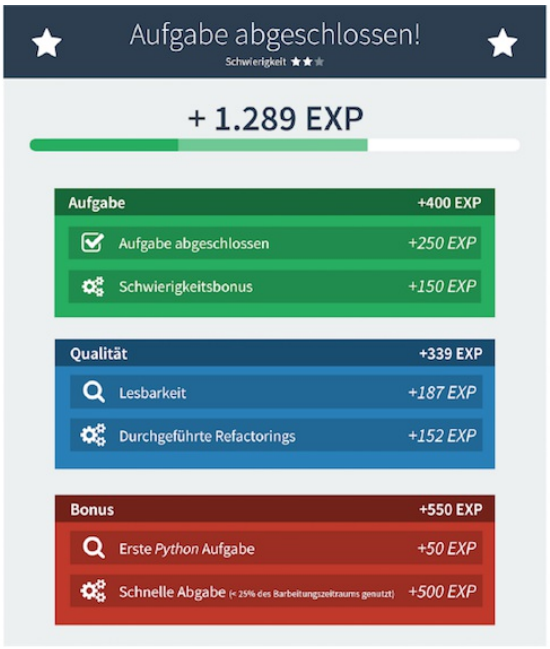


Abbildung A.9.: Achte Seite des Fragebogens: Evaluation der Erfahrungspunkte [Han17]

Was ist deine Meinung zu Erfahrungspunkten und der Bewertung der Aufgabe *

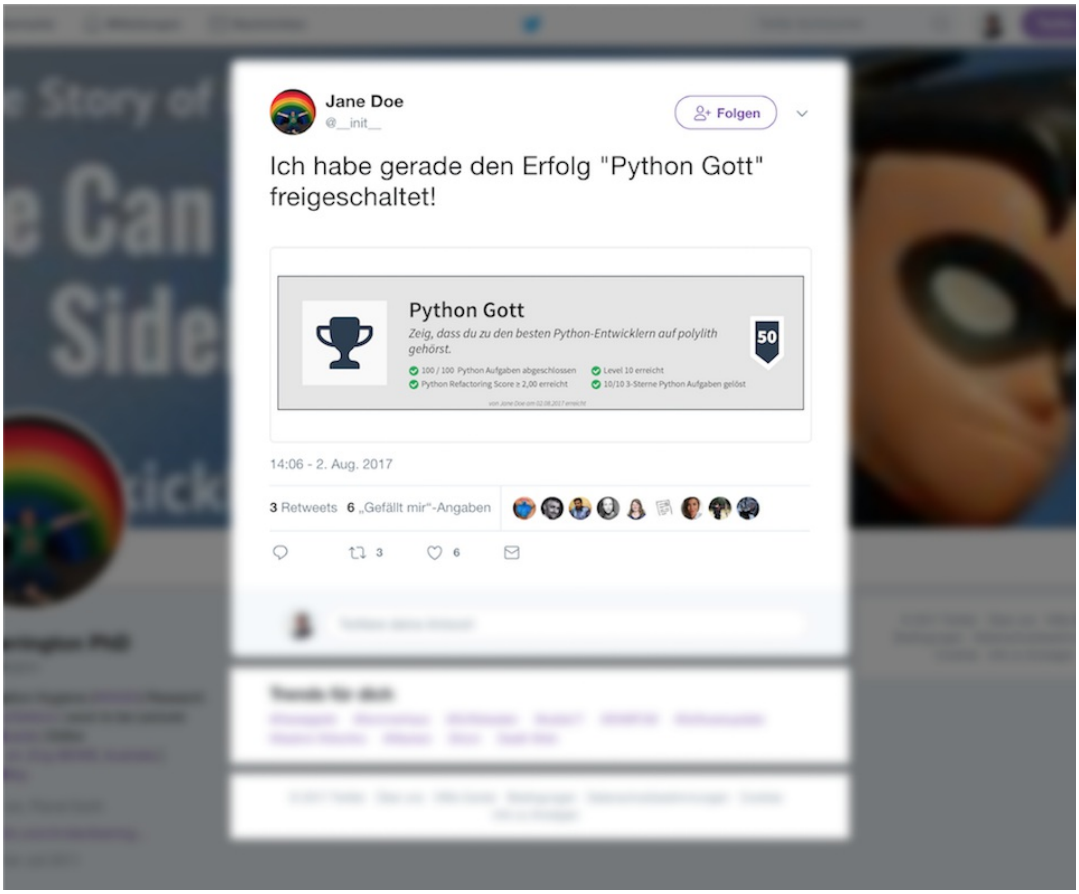
	stimme ich gar nicht zu				stimme ich voll und ganz zu
Ich verstehe was dieses Element aussagt.	◯	◯	◯	◯	◯
Ich finde dieses Element optisch ansprechend.	◯	◯	◯	◯	◯
Ich würde versuchen beim Entwickeln größeren Wert auf die Bewertungskriterien zu legen um mehr Erfahrungspunkte zu erreichen.	◯	◯	◯	◯	◯
Es würde mich demotivieren, wenn meine Freunde mehr Erfahrungspunkte hätten, als ich.	◯	◯	◯	◯	◯
Es würde mich anspornen mehr Erfahrungspunkte zu sammeln als meine Freunde.	◯	◯	◯	◯	◯
Ich habe das Gefühl, dass ich ungerecht bewertet werden könnte.	◯	◯	◯	◯	◯

Kommentar

Abbildung A.10.: Neunte Seite des Fragebogens: Evaluation der Social-Media Integration [Han17]

Seite 6

Social-Media Integration



Erfolge können öffentlich über Social Media Plattformen geteilt werden.

Abbildung A.11.: Zehnte Seite des Fragebogens: Evaluation der Social-Media Integration und Bewertung der Plattform mit Gamification [Han17]

Was ist deine Meinung zu Social Media Integrationen? *

	stimme ich gar nicht zu				stimme ich voll und ganz zu
Ich verstehe was dieser Tweet aussagt.	◯	◯	◯	◯	◯
Es würde mich motivieren denselben Erfolg zu erringen.	◯	◯	◯	◯	◯
Es würde mein Interesse wecken die Plattform zu nutzen.	◯	◯	◯	◯	◯
Es würde mich stören diesen Tweet zu sehen.	◯	◯	◯	◯	◯

Kommentar

Bewertung der Plattform mit den gezeigten Gamification-Elementen

Versuche die Plattform, mit der Erweiterung durch Gamification-Elemente (Erfolge, Erfahrungspunkte, Ranglisten, usw) aus der Sicht des Crowd-Entwicklers anhand der Kriterien des Octalysis-Systems einzuschätzen. Versuche anzugeben wie stark die einzelnen Kriterien ausgeprägt sind.

Zusätzlicher Kontext:
Zusätzlich zur vorher genannten Beschreibung der Plattform, erhalte ich für jede abgeschlossene Aufgabe Erfahrungspunkte und steige Level auf. Am Ende der Aufgabe sehe ich die Übersicht zur Qualität der abgegebenen Aufgabe, und erhalte abhängig davon zusätzliche Erfahrung. (wie im Beispiel Seite 5) Für einige besondere Leistungen (Leistung über einem Grenzwert z.B. besonders gute Qualität, erste Aufgabe in einer Programmiersprache oder Framework, etc.) erhalte ich Erfolge. (wie im Beispiel Seite 3) Diese Erfolge kann ich auf Social-Media Plattformen wie Twitter oder Facebook teilen. Zusätzlich kann ich mich in einer Rangliste mit anderen Nutzern auf der Plattform vergleichen, Ränge aufsteigen, aber auch wieder absteigen.

Epische Bedeutung und Berufung *

Es vermittelt dem Nutzer, dass er an etwas größerem teilnimmt oder das er berufen ist zu einer Sache, die nur er lösen kann.

wenig ausgeprägt stark ausgeprägt

[h!]

Abbildung A.12.: Elfte Seite des Fragebogens: Bewertung der Plattform mit Gamification [Han17]

Entwicklung und Leistung *

Dies ist der innere Antrieb, den jeder Nutzer hat, um voran zu kommen und sich zu entwickeln.

wenig ausgeprägt stark ausgeprägt

Anregung von Kreativität und Feedback *

Wird der Nutzer angeregt seine Kreativität auszuleben, neues auszuprobieren und erhält er dazu sinnvolles Feedback?

wenig ausgeprägt stark ausgeprägt

Eigentum und Besitz *

Der Nutzer möchte gerne Dinge haben und darüber entscheiden können. Das kann sowohl reale Dinge betreffen wie Geld oder Gegenstände, aber auch virtuelle Güter wie Währungen, Punkte, etc.

wenig ausgeprägt stark ausgeprägt

Sozialer Einfluss und Verbundenheit mit anderen Nutzern *

Hierbei geht es darum wie sehr ein Nutzer mit anderen interagiert. Beispielsweise, dass soziale Elemente hergestellt werden, wie. soziale Akzeptanz oder Rückmeldung, aber auch Wettbewerbe.

wenig ausgeprägt stark ausgeprägt

Knappheit und Ungeduld *

Ist der Antrieb etwas haben zu wollen, weil es sehr selten, exklusiv oder gegenwärtig nicht verfügbar ist.

Bsp. einen Gegenstand, einen Status

wenig ausgeprägt stark ausgeprägt

Unberechenbarkeit und Neugier *

Unverhersehbarkeit von Ereignissen, welche den Nutzer betreffen und die Neugierde beim Versuch genau solche Ereignisse auszulösen.

wenig ausgeprägt stark ausgeprägt

Abbildung A.13.: Zwölfte Seite des Fragebogens: Bewertung der Plattform mit Gamification [Han17]

Verlust und Vermeidung *

Die Nutzer wollen ungern etwas verlieren, daher werden sie angetrieben dies zu vermeiden und agieren daher.

Bsp. der Verlust eines Gegenstandes, eines Status oder die Vermeidung von anderen negativen Elementen

wenig ausgeprägt stark ausgeprägt

Kommentar

Seite 8

Alter

Geschlecht

◯ Männlich ◯ Weiblich

Erfahrungen

	keine	wenig	mittel	gute	sehr gute
Programmiererfahrung	◯	◯	◯	◯	◯
Erfahrung mit Entwicklungsumgebungen	◯	◯	◯	◯	◯

Probandennummer *

» **Umleitung auf Schlussseite von Umfrage Online** (ändern)

Abbildung A.14.: Dreizehnte Seite des Fragebogens: Allgemeine Daten

B. EVALUATIONSERGEBNISSE

Im Folgenden werden die Ergebnisse der Nutzerstudie präsentiert.

	Pycharm	Editor	Editor+	Editor+
Proband 1	FizzBuzz	Semiplan	Book tickets for flights	FizzBuzz
Proband 2	FizzBuzz	Book tickets for flights	Semiplan	-
Proband 3	Book tickets for flights	FizzBuzz	Semiplan	-
Proband 4	Semiplan	FizzBuzz	Book tickets for flights	-
Proband 5	FizBuzz	Booking tickets for flights	Semiplan	FizzBuzz
Proband 6	Book tickets for flights	FizzBuzz	Semiplan	-
Proband 7	Semiplan	FizzBuzz	Book tickets for flights	-

Tabelle B.1.: Durchgeführte Testszenarios

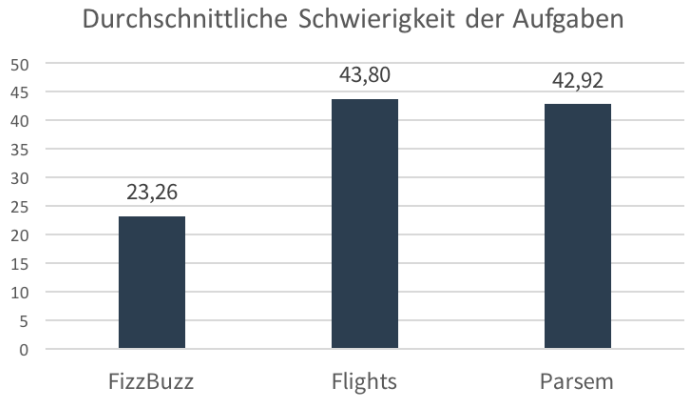


Abbildung B.1.: Durchschnittliche Schwierigkeit der Aufgaben

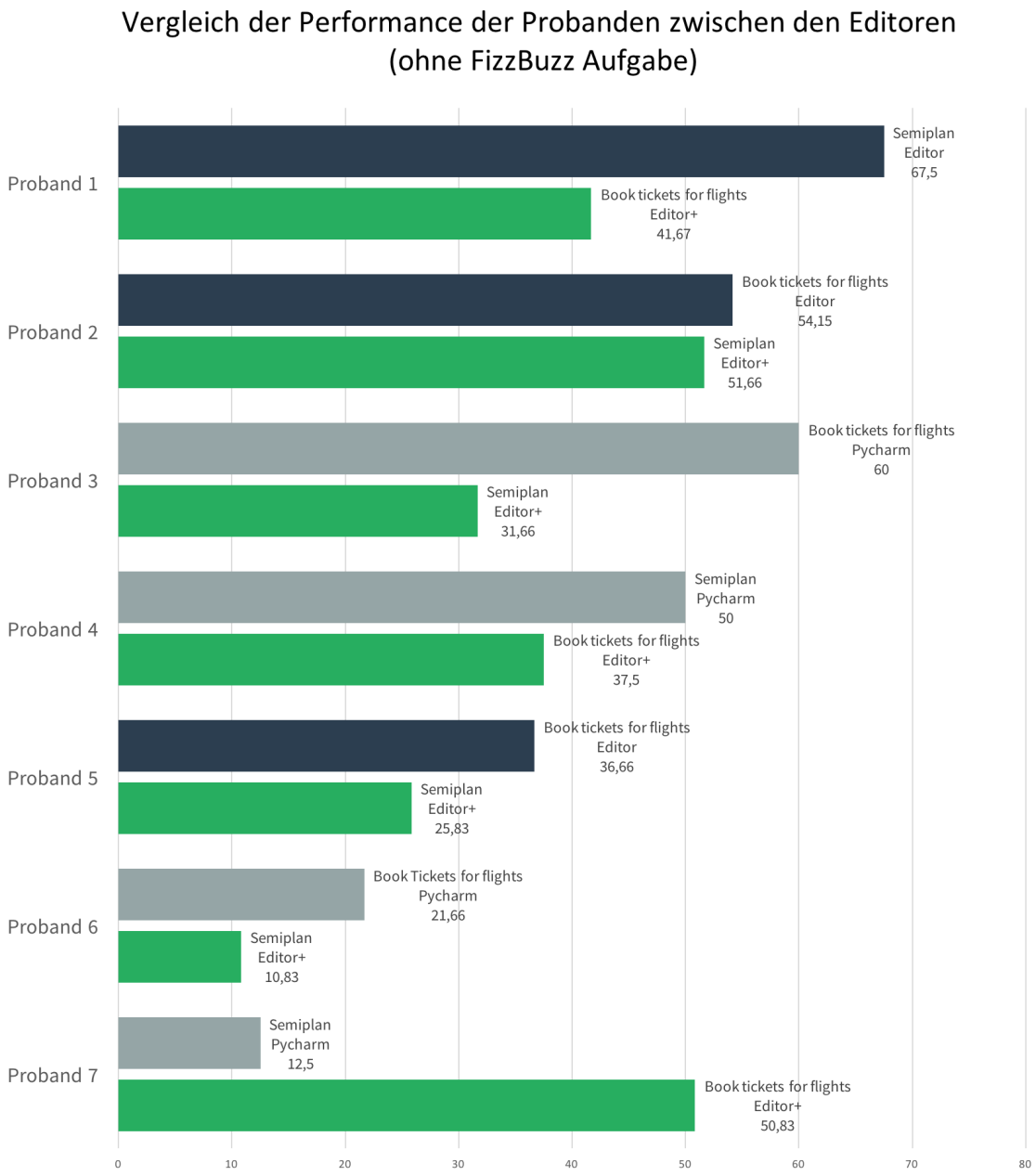


Abbildung B.2.: Vergleich der Editor Performance (Aufgaben mit FizzBuzz wurden aufgrund der Einfachheit der Aufgabe nicht betrachtet)

B.1. PROBAND 1

Alter: 27

Geschlecht: Männlich

Programmiererfahrung: 5 (sehr gute)

Erfahrung mit Entwicklungsumgegbungen: 5 (sehr gute)

TLX Ergebnisse:

	Pycharm FizzBuzz	Editor Semiplan	Editor+ Book tickets for flights	Editor+ FizzBuzz
Geistige Anforderung	10	95	30	15
Körperliche Anforderung	15	95	45	15
Zeitliche Anforderung	50	50	50	10
Leistung	15	50	25	15
Anstrengung	20	20	45	20
Frustration	5	95	55	10
Gesamtbeanspruchung	25,83	67,5	41,66	14,16

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert					X
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert					X
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln			X		

Anmerkungen des Probanden:

- Damit ich den Editor wirklich benutzen würde, müssten noch Refactoring-Funktionen wie Variable auslagern dabei sein und über Tastenkombinationen steuerbar sein

Anmerkungen des Evaluationsleiter:

- empfand die Step Beschreibungen als undeutlich
- verwendete Browser-Suche aber nicht Editor-Suche
- verwendete die Hover-Funktion nicht
- Scrollen der Szenario-Liste war nicht sofort erkennbar
- wünscht sich automatisches Einrücken nach Elementen wie *for* oder *if*

B.2. PROBAND 2

Alter: 26

Geschlecht: Männlich

Programmiererfahrung: 5 (sehr gute)

Erfahrung mit Entwicklungsumgebungen: 5 (sehr gute)

TLX Ergebnisse:

	Pycharm FizzBuzz	Editor Book tickets for flights	Editor+ Semiplan
Geistige Anforderung	55	75	30
Körperliche Anforderung	55	50	55
Zeitliche Anforderung	75	55	55
Leistung	5	5	5
Anstrengung	60	75	55
Frustration	55	65	95
Gesamtbeanspruchung	50,83	54,15	51,66

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert				X	
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert				X	
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln				X	

Anmerkungen des Probanden:

- Hover Hilfen deutlich anzeigen
- Ausgeblendete Code Inhalte mit "..." hervorheben
- Aufgabenbeschreibung deutlicher

Anmerkungen des Evaluationsleiter:

- empfand die Step Beschreibungen als undeutlich
- verwendete Browser-Suche aber nicht Editor-Suche
- verwendete die Hover-Funktion nicht
- Editor$^+$ abgestürzt führte zu erhöhter Frustration

B.3. PROBAND 3

Alter: 30

Geschlecht: Männlich

Programmiererfahrung: 4 (gute)

Erfahrung mit Entwicklungsumgebungen: 4 (gute)

TLX Ergebnisse:

	Pycharm Book tickets for flights	Editor FizzBuzz	Editor+ Semiplan
Geistige Anforderung	90	20	40
Körperliche Anforderung	10	5	5
Zeitliche Anforderung	60	30	30
Leistung	60	50	35
Anstrengung	75	20	55
Frustration	65	40	25
Gesamtbeanspruchung	60	27,5	31,66

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert				X	
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert			X		
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln				X	

Anmerkungen des Probanden:

- Ausklappbare Tests besser hervorheben
- Ladeicon (wenn Tests ausgeführt werden) besser anordnen neben Run-Test Button
- Quelltext auf volle Bildschirmbreite (Szenarien über oder unter Editor oder einklappbar machen)
- Zahlen und Semiplan-Beispiel zu ähnlich

Anmerkungen des Evaluationsleiter:

- schwerer Einstieg in die erste Aufgabe mit Pycharm
- Testausgabe unklar
- fragt häufig nach Instanz oder Datenbank
- ständiges hin-und-her-springen zwischen Test und Quelltext
- lässt den Platzhalter "pass" in der Funktion
- Modul Timezone nicht gefunden
- bei FizzBuzz führt er die Tests nicht aus
- Ausklappen der Tests wird nicht erkannt
- wünscht sich automatisches Einrücken nach Elementen wie *for* oder *if*

B.4. PROBAND 4

Alter: 25

Geschlecht: Männlich

Programmiererfahrung: 4 (gute)

Erfahrung mit Entwicklungsumgebungen: 4 (gute)

TLX Ergebnisse:

	Pycharm Semiplan	Editor FizzBuzz	Editor+ Book tickets for flights
Geistige Anforderung	40	15	60
Körperliche Anforderung	80	15	20
Zeitliche Anforderung	40	15	30
Leistung	20	80	30
Anstrengung	40	15	55
Frustration	80	15	30
Gesamtbeanspruchung	50	25,83	37,5

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert				X	
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert				X	
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln			X		

Anmerkungen des Probanden:

- Offensichtlicher auf Funktionen Hinweisen
- Mehr Feedback bei Testausführung, sowohl automatisch als auch manuell
- Einzelne Tastatureingaben verbessern (Löschen von Zeile funktionierte nur über Umwege)

Anmerkungen des Evaluationsleiter:

- durchgehend gute Performance des Probanden
- erster Proband, der die automatische Testausführung aktiviert
- Probleme beim Einfügen von Leerspalten nach der zu implementierenden Funktion
- fühlt sich unsicher im Editor (kein Vertrauen in den Editor)
- unklarer Fehler, da Autovervollständigung bei Funktionen nicht automatisch Klammern setzt

B.5. PROBAND 5

Alter: 31

Geschlecht: Männlich

Programmiererfahrung: 3 (mittel)

Erfahrung mit Entwicklungsumgebungen: 3 (mittel)

TLX Ergebnisse:

	Pycharm FizzBuzz	Editor Book tickets for flights	Editor+ Semiplan	Editor+ FizzBuzz
Geistige Anforderung	15	55	30	10
Körperliche Anforderung	5	15	10	10
Zeitliche Anforderung	10	40	20	20
Leistung	20	35	35	10
Anstrengung	25	45	30	10
Frustration	5	30	30	15
Gesamtbeanspruchung	13,3	36,66	25,83	12,5

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert					X
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert					X
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln			X		

Anmerkungen des Probanden:

- Wunsch nach dunklem Theme

Anmerkungen des Evaluationsleiter:

- keine Erfahrung mit Django und wenig Erfahrung mit objektorientierter Programmierung
- trotz fehlender Erfahrung sehr gute Performance
- nutzt in Pycharm häufig die Autovervollständigung
- wünscht sich automatisches Einrücken nach Elementen wie *for* oder *if*
- traute dem Editor$^+$ keine automatische Vervollständigung zu, da Vorschläge erst nach Tastenkombination angezeigt werden
- bei Pycharm werden Vorschläge automatisch angezeigt

B.6. PROBAND 6

Alter: 31

Geschlecht: Männlich

Programmiererfahrung: 5 (sehr gute)

Erfahrung mit Entwicklungsumgebungen: 5 (sehr gute)

TLX Ergebnisse:

	Pycharm Book tickets for flights	Editor FizzBuzz	Editor+ Semiplan
Geistige Anforderung	25	10	20
Körperliche Anforderung	15	10	5
Zeitliche Anforderung	20	5	10
Leistung	40	5	5
Anstrengung	25	10	20
Frustration	5	5	5
Gesamtbeanspruchung	21,66	7,5	10,83

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert				X	
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert				X	
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln				X	

Anmerkungen des Probanden:

- Ausklappen der Tests auf den ersten Blick nicht wahrnehmbar

Anmerkungen des Evaluationsleiter:

- erfahrener Entwickler (Django und Python)
- führt sofort die Tests aus
- benutzt zunächst nicht die `create_ticket_method`, sondern implementiert eigene Funktion
- findet sich in Pycharm Testübersicht nicht zurecht
- Ausklappen der Tests nicht erkannt (Editor+)
- öffnet Anwendung noch einmal um Tests wieder grau zu bekommen
- verwendet bei Aufgabe im Editor die automatische Testausführung

B.7. PROBAND 7

Alter: 31

Geschlecht: Männlich

Programmiererfahrung: 5 (sehr gute)

Erfahrung mit Entwicklungsumgebungen: 4 (gute)

TLX Ergebnisse:

	Pycharm Semiplan	Editor FizzBuzz	Editor+ Book tickets for flights
Geistige Anforderung	30	80	80
Körperliche Anforderung	5	5	5
Zeitliche Anforderung	5	55	75
Leistung	5	30	40
Anstrengung	25	75	80
Frustration	5	25	25
Gesamtbeanspruchung	12,5	45	50,83

Bewertung des Editors mit Hilfen (Editor+):

	stimme ich gar nicht zu				stimme ich voll und ganz zu
	1	2	3	4	5
Ich verstehe, wie der Editor funktioniert					X
Ich fand es hilfreich, dass irrelevante Klassen ausgeblendet werden					X
Der Editor hat mir die Arbeit erleichtert					X
Ich würde den Editor nutzen, um kleine Aufgaben zu entwickeln					X

Anmerkungen des Probanden:

- Es war nicht unmittelbar sichtbar, dass man die Testbeschreibungen ausklappen kann

Anmerkungen des Evaluationsleiter:

- erfahrener Entwickler (Django und Python)
- zunächst Suche in Pycharm (Semiplan Aufgabe)
- Aufgabe in Pycharm sehr schnell gelöst
- Ausklappen der Tests nicht erkannt (Editor+)